APPROVAL SHEET

Title of Dissertation: DEVELOPING A COMPUTATIONAL MODEL OF NEURAL NETWORKS INTO A LEARNING MACHINE

Name of Candidate: Bryce Mackey-Williams Carey
Doctor of Philosophy, 2017

Dissertation and Abstract Approved: _____

Dr. James Lo
Professor
Department of Mathematics & Statistics

Date Approved: _____

ABSTRACT

Title of dissertation: DEVELOPING A COMPUTATIONAL MODEL OF NEURAL NETWORKS INTO A LEARNING MACHINE

Bryce Mackey-Williams Carey,
Doctor of Philosophy, 2017

Dissertation directed by: Professor James Lo
Department of Mathematics & Statistics

The purpose of this dissertation work is to contribute to the development of a biologically plausible model of neural networks into a learning machine. Temporal hierarchical probabilistic associative memory (THPAM) is a functional model of biological neural networks which performs a variant of supervised and unsupervised Hebbian learning to store information in the synapses, uses dendritic trees to encode information, and communicates information via spike trains. THPAM can be viewed as a recurrent hierarchical network of processing units, neuronal compartments that serve as pattern recognizers. This work proposes supplemental developments, a parallel programming implementation, and several benchmark results pertaining to the processing unit architecture.

Supplemental theories and mechanisms pertaining to the processing unit architecture are contributed in this dissertation. These contributions serve to confirm propositions contained in original publications, enable alternative constructions of the processing unit generalization component, and allow for an alternative general-

ization mechanism. The new generalization mechanism has a unique application in efficiently learning data clusters centered at a target input vector.

Orthogonal expansion of a vector in the processing unit is an exponential function of the dimension of the vector. Although there are ways to avoid vectors with a large dimension, a parallel programming implementation proposed in this work is utilized to somewhat alleviate the severe limitations imposed by this complexity on serial machines. The scalability of the parallel program is examined on the maya cluster of the UMBC High Performance Computing Facility. The parallelized processing unit implementation is beneficial in reducing the run time of sufficiently large fixed problem sizes from several hours to a few seconds.

The performance of the processing unit as a pattern recognizer is demonstrated on sample data sets obtained from the UCI Machine Learning Repository. These data sets independently contained categorical data, missing data, and real-valued data. Several data encoding techniques are performed and examined in order to best suit the predictive performance of the processing unit on the data sets considered. Differences in performance between particular encoding methods are thoroughly examined and discussed in relation to the processing unit mechanisms, and the effects hyperparameter adjustments are precisely considered.

DEVELOPING A COMPUTATIONAL MODEL OF NEURAL NETWORKS INTO A LEARNING MACHINE

by

Bryce Mackey-Williams Carey

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Advisory Committee:
Professor James Lo, Chair/Advisor
Professor Tim Finin
Professor Jacob Kogan
Professor Tim Oates
Professor Junyong Park

Dedication

I dedicate this dissertation to my family, whom without their love and support this would not have been possible. Thank you for your encouragement, advice, and your hospitality throughout my years of education. Thank you for your patience with me, for lending your ears during difficult times, and for celebrating in times of success. I think of you always.

I also dedicate this dissertation to my fiancée, Hyekyung Park, with whom I shared many of the same trials and tribulations as doctoral students of applied mathematics. With you, I have witnessed more of the world than I ever hoped. I am forever grateful to be in a universe where we had the opportunity to meet, and I eagerly await becoming your husband and experiencing the rest of our lives together.

Acknowledgments

First and foremost I thank my advisor, Professor James Ting-Ho Lo, for giving me the invaluable opportunity to perform this dissertation research pertaining to his biological model of neural networks. He has always made himself available to provide timely help and advice, whether in person or remotely. Through him I gained interest in the fields of machine learning and artificial intelligence, satisfying my desire to combine my interests in mathematics and computer science. I look forward to future collaboration with him and his students on his research.

I thank my committee members, Professor Tim Finin, Professor Jacob Kogan, Professor Tim Oates, and Professor Junyong Park, for agreeing to serve on my dissertation committee.

I thank the faculty and staff of the Department of Mathematics and Statistics at UMBC for their instruction, support, and assistance over the years I've been in the program. The present and former graduate program directors, Professor Animikh Biswas, Professor Muruhan Rathinam, and Professor Kathleen Hoffman, provided vital program advice and support for myself and my peers. I value the leadership and counsel of the current and former department chairs, Professor Rouben Rostamian and Professor Nagaraj Neerchal. Senior Lecturer Bonnie Tighe instilled in me a penchant for teaching, and I will always value her guidance to graduate students on effectively fulfilling the roles of teaching assistant and instructor. Department Coordinator Janet Burgee and Department Manager Margaret Kennedy worked tirelessly to maintain the administrative aspect of the department, and I

could always depend on them for scheduling arrangements and teaching assignments. Additionally, I thank all of the teachers I had the immense pleasure to learn from throughout my graduate student career at UMBC.

I acknowledge the financial support I received from the Department of Mathematics and Statistics at UMBC, which enabled me to pursue graduate study full-time over the last 6 years.

Finally, I thank my fellow graduate students I befriended over the years, within and without the applied mathematics program. One of the most significant lessons I learned from you all is that it is far better to experience hardships and rejoice in celebration among friends. Our camaraderie was a substantial factor which kept me pressing forward from beginning to end.

Thank you all.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ANN	Artificial Neural Network
HPCF	High Performance Computing Facility
MPI	Message Passing Interface
UCI	University of California, Irvine
UMBC	University of Maryland, Baltimore County
THPAM	Temporal Hierarchical Probabilistic Associative Memory

Chapter 1

Introduction

1.1 Motivation

Backpropagation based artificial neural networks (ANNs), namely multilayer perceptrons, convolutional neural networks, and deep neural networks, are at present the leading technologies in machine learning and artificial intelligence research and application, commonly used as the most reliable tools for pattern recognition, classification, function approximation, and signal processing. These ANNs can be adapted to learn and perform a variety of specific tasks via its loose imitation of a functional cluster of biological neurons connected by axons and dendrites with modifiable strengths or weights. These weights are adjusted to reduce an overall performance error in the output of the neural network architecture by means of iteration and back propagation, among other customized procedures, involving sophisticated mathematical techniques from differentiation and optimization.

Although backpropagation based ANNs are a valuable and mature technology with numerous applications, there is an discernible lack of evidence in support of a biological equivalent to back propagation within natural neural networks. Biological neural networks serve as inspiration in their development, but the objective of these ANNs is not necessarily to replicate nature but rather to serve as engineered utilities suitable for solving classes of problems which were once considered intractable.

Temporal hierarchical probabilistic associative memory (THPAM) is a functional model of biological neural networks which is meant to retain biological plausibility as natural neural networks are presently understood. The model features entirely automated learning mechanisms without the use of iterative data cycling, differentiation, or optimization. Except for the mechanism of encoding inputs by dendrites, the mechanisms in THPAM are justified with findings reported in biological literature. This model may serve as an initial attempt to faithfully replicate the functions of natural neural networks, and further biological findings may substantiate THPAM and offer opportunities in its future development to fully capture the functionality of the biological brain.

This dissertation research is motivated by the established theory pertaining to the processing unit architecture, a pattern recognizer which is postulated to comprise a recurrent hierarchical network of processing units that constitute THPAM. This work contains additional theoretical contributions toward the development of the processing unit, provides an original parallelized implementation of the processing unit architecture, and establishes initial benchmarks involving the application of the processing unit towards sample data sets. These efforts in understanding, implementing, and testing the processing unit will serve as a foundation in further establishing the theory of THPAM and its eventual realization as a coherent learning machine.

1.2 Thesis Statement

This dissertation proposes several theoretical additions promoting the research and development of the THPAM processing unit. Additionally, this work provides a parallelized implementation of the processing unit architecture, which is integral in the application of the processing unit towards data sets. Finally, this paper reports experimental results of the processing unit applied to sample data sets comprised of different data types in order to provide initial benchmarks, explanations, and suggestions for best practices in future work.

1.3 Contributions

The main contributions of this dissertation are the following:

- Demonstrated that the following property holds for ternary vectors $\mathbf{x}$ and $\mathbf{z}$ of the same dimension, with the orthogonal expansion function Φ: $\Phi(\mathbf{x} \circ \mathbf{z}) = \Phi(\mathbf{x}) \circ \Phi(\mathbf{z})$, where $\circ$ denotes the entrywise product. See Theorem 3 in Section 3.2.

- Developed an efficient method of constructing particular masking matrices, which are used in generalization. See equations 3.1 and 3.2 in Sections 3.3 and 3.4, respectively.

- Proposed an alternative generalization scheme with entry flipping, which can additionally be used to efficiently learn data clusters within a specified Hamming distance from a central point. See Section 3.4.

- Developed a parallelized implementation of the processing unit architecture and examined its scalability with varying problem size and number of parallel processes used. See Chapter 4.

- Produced initial performance benchmarks of the processing unit applied to different data types with several input encodings, and provided explanations and suggestions for best practices. See Chapter 5.

1.4 Thesis Outline

The dissertation is organized in the following manner. Chapter 2 introduces the relevant background material necessary to perform this research, namely the architecture and machinations of the THPAM processing unit. Chapter 3 presents additional theory submitted as original contributions in this work. Chapter 4 discusses a parallelized version of the processing unit implementation and examines its scalability as the problem size and number of parallel processes vary. Chapter 5 reports the results of several experiments involving the application of the processing unit towards different types of data with different data encoding schemes. Chapter 6 concludes the dissertation and outlines opportunities for future work.

Chapter 2

Background and Related Work

2.1 Overview

This chapter summarizes the theory of the THPAM processing unit architecture as it relates to the work conducted for this dissertation. Few original publications exist describing more deeply the processing unit and its role in the overall model, mechanisms which remain unused in this work, and propositions regarding the hierarchical structure of processing units comprising THPAM [13, 14, 15]. As such, selected theories are explained thoroughly by examples or proofs in order to compose a coherent document, but finer reasonings should be sought in the original publications. None of the material presented in this chapter is claimed to be original work as part of this dissertation submission, and all original contributions discussed in the subsequent chapters are based on the material provided herein. Some notational differences are employed in order to present this background material in a more accessible manner for the intended audience.

Section 2.2 describes the processing unit structure, its internal representation for input vectors, and its procedures for storing learned information. Section 2.3 describes the procedure of extracting stored information from the processing unit structure in order to generate output predictions on target inputs. The generalization mechanism involving entry masking is also discussed in detail.

2.2 Processing Unit Structure

2.2.1 Orthogonal Expansion

The following definition is provided in order to avoid ambiguity in using particular terms throughout this dissertation.

Definition 1.

- A *binary* vector is one whose entries all belong to the set $\{0, 1\}$.
- A *bipolar* vector is one whose entries all belong to the set $\{-1, 1\}$.
- A *ternary* vector is one whose entries all belong to the set $\{-1, 0, 1\}$.

Note that binary and bipolar vectors are also ternary vectors. THPAM primarily involves interactions between orthogonal expansions of ternary vectors from coding theory [22]. As the name suggests, the orthogonal expansion is a ternary vector encoding designed to induce orthogonality between vectors which differ in a particular manner. To introduce this concept, we first visit an example involving the orthogonal expansion of bipolar vectors.

Example 1. *Let* $\mathbf{x} = (\,x_1\ x_2\,)^T$ *and* $\mathbf{z} = (\,z_1\ z_2\,)^T$ *be bipolar vectors. Construct orthogonal expansions* $\Phi(\mathbf{x})$ *and* $\Phi(\mathbf{z})$ *of* $\mathbf{x}$ *and* $\mathbf{z}$*, respectively, to be*

$$\Phi(\mathbf{x}) = \begin{pmatrix} 1 & x_1 & x_2 & x_1x_2 \end{pmatrix}^T$$
$$\Phi(\mathbf{z}) = \begin{pmatrix} 1 & z_1 & z_2 & z_1z_2 \end{pmatrix}^T$$

Then

$$(\Phi(\mathbf{x}))^T\Phi(\mathbf{z}) = 1 + x_1z_1 + x_2z_2 + x_1x_2z_1z_2 = (1 + x_1z_1)(1 + x_2z_2).$$

Note that $(\Phi(\mathbf{x}))^T\Phi(\mathbf{x}) = 2^2 = 4$. *If* $\mathbf{x} \neq \mathbf{z}$, *then there exists* x_i *and* z_i *such that* $x_i \neq z_i$ *for some* $i = 1, 2$. *Since* $\mathbf{x}$ *and* $\mathbf{z}$ *are bipolar, it must be the case that* $x_i = -z_i$, *so* $1 + x_i z_i = 1 - x_i x_i = 1 - 1 = 0$, *resulting in* $(\Phi(\mathbf{x}))^T\Phi(\mathbf{z}) = 0$. *Note that this orthogonality between* $\Phi(\mathbf{x})$ *and* $\Phi(\mathbf{z})$ *is not guaranteed if either vector is permitted to be ternary, since entries consisting of zero in either vector do not cause the inner product to be zero.*

The following is the general definition of the orthogonal expansion for ternary vectors.

Definition 2 (Orthogonal Expansion)**.** Let $\mathbf{x} = (\, x_1\ x_2\ \cdots\ x_n\,)^T$ be a ternary vector. The *orthogonal expansion* of $\mathbf{x}$, denoted $\Phi(\mathbf{x})$ where $\Phi : \{-1, 0, 1\}^n \to \{-1, 0, 1\}^{2^n}$, is produced recursively as follows:

$$
\begin{aligned}
\Phi_1(\mathbf{x}) &= \begin{pmatrix} 1 \\ x_1 \end{pmatrix}, \\
\Phi_2(\mathbf{x}) &= \begin{pmatrix} \Phi_1(\mathbf{x}) \\ x_2\Phi_1(\mathbf{x}) \end{pmatrix}, \\
&\vdots \\
\Phi_i(\mathbf{x}) &= \begin{pmatrix} \Phi_{i-1}(\mathbf{x}) \\ x_i\Phi_{i-1}(\mathbf{x}) \end{pmatrix}, \\
&\vdots \\
\Phi(\mathbf{x}) = \Phi_n(\mathbf{x}) &= \begin{pmatrix} \Phi_{n-1}(\mathbf{x}) \\ x_n\Phi_{n-1}(\mathbf{x}) \end{pmatrix}.
\end{aligned}
\tag{2.1}
$$

Producing the orthogonal expansion has algorithmic complexity $O(2^n)$, which is intractable even for modest values of input dimension n. This curse of dimensionality imposes a severe limitation on the practicality of a single processing unit. Although this dissertation focuses on the application of a single processing unit, THPAM proposes that multiple processing units are assigned to observe subsets of the input, effectively alleviating this dimensionality issue.

The subsequent theorem summarizes important properties regarding the inner products of orthogonal expansions.

Theorem 1. *Let* $\mathbf{a} = (\, a_1 \; a_2 \; \cdots \; a_n \,)^T$ *and* $\mathbf{b} = (\, b_1 \; b_2 \; \cdots \; b_n \,)^T$ *be ternary vectors. The inner product of their orthogonal expansions can be expressed as*

$$(\Phi(\mathbf{a}))^T \Phi(\mathbf{b}) = \prod_{i=1}^{n} (1 + a_i b_i),$$

which has the following properties:

1. *If* $a_k b_k = -1$ *for some* $k \in \{1, \ldots, n\}$, *then* $(\Phi(\mathbf{a}))^T \Phi(\mathbf{b}) = 0$,
2. *If* $a_k b_k = 0$ *for some* $k \in \{1, \ldots, n\}$, *then* $(\Phi(\mathbf{a}))^T \Phi(\mathbf{b}) = \prod_{i=1, i \neq k}^{n} (1 + a_i b_i)$,
3. *If* $(\Phi(\mathbf{a}))^T \Phi(\mathbf{b}) \neq 0$, *then* $(\Phi(\mathbf{a}))^T \Phi(\mathbf{b}) = 2^{\mathbf{a}^T \mathbf{b}}$,
4. *If* $\mathbf{a}$ *and* $\mathbf{b}$ *are bipolar, then* $(\Phi(\mathbf{a}))^T \Phi(\mathbf{b}) = 0$ *if* $\mathbf{a} \neq \mathbf{b}$, *and* $(\Phi(\mathbf{a}))^T \Phi(\mathbf{b}) = 2^n$ *if* $\mathbf{a} = \mathbf{b}$.

Proof. A proof by induction can be used to demonstrate that for all $j \in \{1, \ldots, n\}$,

$$(\Phi_j(\mathbf{a}))^T \Phi_j(\mathbf{b}) = \prod_{i=1}^{j} (1 + a_j b_j).$$

(Base Step): Clearly, $(\Phi_1(\mathbf{a}))^T \Phi_1(\mathbf{b}) = 1 + a_1 b_1$.

(Inductive Step): Suppose $(\Phi_j(\mathbf{a}))^T \Phi_j(\mathbf{b}) = \prod_{i=1}^{j}(1 + a_i b_i)$ for $j \in \{1, \ldots, n-1\}$. Then by Definition 2,

$$\begin{aligned}
(\Phi_{j+1}(\mathbf{a}))^T \Phi_{j+1}(\mathbf{b}) &= \begin{pmatrix} \Phi_j(\mathbf{a}) \\ a_{j+1}\Phi_j(\mathbf{a}) \end{pmatrix}^T \begin{pmatrix} \Phi_j(\mathbf{b}) \\ b_{j+1}\Phi_j(\mathbf{b}) \end{pmatrix} \\
&= (\Phi_j(\mathbf{a}))^T \Phi_j(\mathbf{b}) + a_{j+1} b_{j+1} (\Phi_j(\mathbf{a}))^T \Phi_j(\mathbf{b})) \\
&= (1 + a_{j+1} b_{j+1})(\Phi_j(\mathbf{a}))^T \Phi_j(\mathbf{b}) \\
&= \prod_{i=1}^{j+1}(1 + a_i b_i)
\end{aligned}$$

The properties follow via simple inspection. □

In summary, two expansions are orthogonal if their base vectors are bipolar and unequal. More generally, the expansions are orthogonal if their base vectors are ternary and at least one pair of corresponding entries produces a negative product. An entry of zero in either vector effectively ignores the corresponding entry in the other vector, and the inner product of the orthogonal expansions is computed as if that entry was never present. If two orthogonal expansions are not orthogonal, then their inner product produces a power of two with the exponent equal to the inner product of the base vectors.

2.2.2 Correlation Learning

The processing unit learns in a manner akin to the Hebbian learning rule [7] as defined below.

Definition 3. A *processing unit* is a pair $(\mathbf{D}, \mathbf{C})_{n,m}$ with $\mathbf{D}$ of dimension $m \times 2^n$ and $\mathbf{C}$ of dimension 2^n constructed according to the following formulas on bipolar pairs $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\}$, with $\mathbf{x}$ n-dimensional and $\mathbf{y}$ m-dimensional:

$$\mathbf{D} = \sum_{i=1}^{N} \mathbf{y}_i (\Phi(\mathbf{x}_i))^T,$$
$$\mathbf{C} = \sum_{i=1}^{N} (\Phi(\mathbf{x}_i))^T.$$

Matrix $\mathbf{D}$ and vector $\mathbf{C}$ are called the *expansion correlation matrices*. Particularly, the processing unit is updated according to the following learning rule on bipolar pair $(\mathbf{x}, \mathbf{y})$:

$$\mathbf{D} \leftarrow \mathbf{D} + \mathbf{y}(\Phi(\mathbf{x}))^T,$$
$$\mathbf{C} \leftarrow \mathbf{C} + (\Phi(\mathbf{x}))^T.$$

Matrix $\mathbf{D}$ stores the outer products of the input orthogonal expansions with their corresponding labels, and vector $\mathbf{C}$ stores the input orthogonal expansions. The utility of this storage scheme is presented in the next section.

2.3 Prediction and Generalization

2.3.1 Empirical Probability

The following theorem demonstrates a method of extracting information from a processing unit in order to predict the label of a target input vector.

Theorem 2. *Let* $(\mathbf{D}, \mathbf{C})_{n,m}$ *be a processing unit that has performed covariance and accumulation learning on the bipolar pairs* $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\}$. *Let* $\mathbf{z}$

be an n-dimensional bipolar vector. If $\mathbf{C}\Phi(\mathbf{z}) \neq 0$ then

$$\frac{\mathbf{D}\Phi(\mathbf{z})}{\mathbf{C}\Phi(\mathbf{z})} = \frac{\sum_{i=1,\, \mathbf{x}_i=\mathbf{z}}^{N} \mathbf{y}_i}{|\{\mathbf{x}_i : \mathbf{x}_i = \mathbf{z}\}|}. \tag{2.2}$$

Proof. By applying Theorem 1,

$$\begin{aligned}
\frac{\mathbf{D}\Phi(\mathbf{z})}{\mathbf{C}\Phi(\mathbf{z})} &= \frac{\sum_{i=1}^{N} \mathbf{y}_i (\Phi(\mathbf{x}_i))^T \Phi(\mathbf{z})}{\sum_{i=1}^{N} (\Phi(\mathbf{x}_i))^T \Phi(\mathbf{z})} \\
&= \frac{2^n \sum_{i=1,\, \mathbf{x}_i=\mathbf{z}}^{N} \mathbf{y}_i}{2^n \sum_{i=1,\, \mathbf{x}_i=\mathbf{z}}^{N} 1} \\
&= \frac{\sum_{i=1,\, \mathbf{x}_i=\mathbf{z}}^{N} \mathbf{y}_i}{|\{\mathbf{x}_i : \mathbf{x}_i = \mathbf{z}\}|}.
\end{aligned}$$

This completes the proof. □

With a target input, the processing unit is capable of producing an average over the labels learned with input vectors equivalent to the target. This average would be an exact label providing that no contradictions are present among the learned input and label pairs. Regardless, each entry in this average will reside within the interval $[-1, 1]$, which can then be converted to a bipolar vector via thresholding techniques. Alternatively, this average can be regarded as a type of probability about whether a label entry is positive or negative for a target input.

Definition 4. Let $(\mathbf{D}, \mathbf{C})_{n,m}$ be a processing unit. The *empirical probability* of n-dimensional bipolar input $\mathbf{z}$, denoted $\rho(\mathbf{z}) : \{-1, 1\}^n \to [0, 1]^m$, is defined as

$$\rho(\mathbf{z}) = \begin{cases} \frac{1}{2}\left(\frac{\mathbf{D}\Phi(\mathbf{z})}{\mathbf{C}\Phi(\mathbf{z})} + \mathbf{e}\right) & \text{if } \mathbf{C}\Phi(\mathbf{z}) \neq 0, \\ \frac{1}{2}\mathbf{e} & \text{if } \mathbf{C}\Phi(\mathbf{z}) = 0, \end{cases} \tag{2.3}$$

where $\mathbf{e}$ denotes the m-dimensional vector of all ones.

The empirical probability is simply an affine transformation of the average label produced in Theorem 2 to a vector whose entries all reside within interval $[0, 1]$. This can be used in combination with a pseudorandom number generator to construct a strictly bipolar label.

A glaring deficiency of this form of the empirical probability construction is that it lacks the ability to generalize. The empirical probability on an unlearned target input will always produce a vector with 0.5 in every entry, meaning no learned information could be used in predicting the output label. This is generally unacceptable in machine learning for good reasons: slight deviations in data should not completely discount the ability of a model to generate an educated prediction, and most practical problems have potentially enormous or infinite feature spaces that cannot be realistically covered by all of the training inputs. Thus, a generalization mechanism is required for the processing unit, which is discussed next.

2.3.2 Masking Matrix

In order for a processing unit to generalize on a target input vector, the inner products between its orthogonal expansion and the learned orthogonal expansions must be altered to produce nonzero results for learned vectors which are similar, but not necessarily equal, to the target vector. The goal of generalization is to produce educated predictions on unlearned inputs. This requires a mechanism to relax the strict orthogonality of the expansions of bipolar vectors. Recall the second and third properties from Theorem 1, which essentially state that any zero entries

in either of two ternary vectors effectively skip both corresponding entries when computing the inner product of their orthogonal expansions. An automatic method of replacing particular entries in target vector with zeros would thereby produce a nontrivial empirical probability vector, providing the processing unit has been sufficiently trained. The next example demonstrates such a method.

Example 2. *Let* $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$ *and* $\mathbf{z} = \begin{pmatrix} z_1 & z_2 & z_3 \end{pmatrix}^T$*, with* $x_1 \neq z_1$*,* $x_2 = z_2$*, and* $x_3 = z_3$*. Then* $(\Phi(\mathbf{x}))^T \Phi(\mathbf{z}) = 0$*, and we wish to alter this inner product to produce a nonzero value by skipping* z_1*. Entry* z_1 *can be removed from* $\mathbf{z}$ *by performing the product*

$$\operatorname{diag}\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 0 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 0 \\ x_2 \\ x_3 \end{pmatrix},$$

then computing the orthogonal expansion of the resulting vector, but this will be costly and inefficient to maintain the original inputs and repeatedly produce these orthogonal expansions when needed. Instead, it can be demonstrated that the above equality is holds between orthogonal expansions, meaning

$$\operatorname{diag}\left(\Phi\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}\right)\Phi\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \Phi\begin{pmatrix} 0 \\ z_2 \\ z_3 \end{pmatrix} = \Phi\begin{pmatrix} 0 \\ x_2 \\ x_3 \end{pmatrix},$$

and by Theorem 1,

$$(\Phi(\mathbf{x}))^T \operatorname{diag}\left(\Phi\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}\right)\Phi(\mathbf{z}) = \left(\Phi\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}\right)^T \Phi\begin{pmatrix} 0 \\ x_2 \\ x_3 \end{pmatrix} = 2^2 = 4,$$

accomplishing the objective without revisiting the original vectors $\mathbf{x}$ *and* $\mathbf{z}$*.*

The following proposition generalizes the process observed in Example 2.

Proposition 1. *Let* $\mathbf{z}$ *be a bipolar vector and let* $\{i_1, i_2, \ldots, i_j\}$ *be a set of indices on* $\mathbf{z}$*. Then*

$$\operatorname{diag}(\Phi(\mathbf{e}_{i_1^0,i_2^0,\ldots,i_j^0}))\Phi(\mathbf{z}) = \Phi(\mathbf{z}_{i_1^0,i_2^0,\ldots,i_j^0}),$$

where $\mathbf{z}_{i_1^0,i_2^0,\ldots,i_j^0}$ *and* $\mathbf{e}_{i_1^0,i_2^0,\ldots,i_j^0}$ *respectively denote vectors* $\mathbf{z}$ *and* $\mathbf{e}$ *with zeros replacing their entries on the specified index set. This procedure of replacing target vector entries with zeros via its orthogonal expansion is called* entry masking*.*

A formal proof of a more general version of this proposition is provided in the next chapter. For now, we visit an example in which the proposition is applied to perform several entry maskings at once.

Example 3. *Let* $\mathbf{z} = \begin{pmatrix} z_1 & z_2 & z_3 \end{pmatrix}^T$*. Then by Proposition 1,*

$$\operatorname{diag}\left(\Phi\begin{pmatrix}1\\1\\1\end{pmatrix} + \Phi\begin{pmatrix}0\\1\\1\end{pmatrix} + \Phi\begin{pmatrix}1\\0\\1\end{pmatrix} + \Phi\begin{pmatrix}1\\1\\0\end{pmatrix}\right)\Phi(\mathbf{z}) =$$

$$\Phi\begin{pmatrix}z_1\\z_2\\z_3\end{pmatrix} + \Phi\begin{pmatrix}0\\z_2\\z_3\end{pmatrix} + \Phi\begin{pmatrix}z_1\\0\\z_3\end{pmatrix} + \Phi\begin{pmatrix}z_1\\z_2\\0\end{pmatrix},$$

producing a sum of $\Phi(\mathbf{z})$ *and all of its entry maskings in which one entry is masked.*

With 3-dimensional bipolar vector $\mathbf{x}$*, this can be used in the bilinear form*

$$(\Phi(\mathbf{x}))^T \text{diag}\left(\Phi\begin{pmatrix}1\\1\\1\end{pmatrix} + \Phi\begin{pmatrix}0\\1\\1\end{pmatrix} + \Phi\begin{pmatrix}1\\0\\1\end{pmatrix} + \Phi\begin{pmatrix}1\\1\\0\end{pmatrix}\right)\Phi(\mathbf{z}),$$

which would be nonzero providing $d(\mathbf{x}, \mathbf{z}) \leq 1$ *in Hamming distance.*

This leads into the generalization mechanism of the processing unit.

Definition 5 (Masking Matrix)**.** A matrix $\mathbf{M}_l$ is a *masking matrix* if it is of the form

$$\mathbf{M}_l = \text{diag}\left(\Phi(\mathbf{e}) + \sum_{j=1}^{l}\sum_{i_j=j}^{n}\cdots\sum_{i_2=2}^{i_3-1}\sum_{i_1=1}^{i_2-1}\alpha_{i_1,i_2,\ldots,i_j}\Phi(\mathbf{e}_{i_1^0,i_2^0,\ldots,i_j^0})\right),$$

where all $\alpha_{i_1,i_2,\ldots,i_j} > 0$ are scalar weights, $\mathbf{e}$ denotes the n-dimensional vector of all ones, and $\mathbf{e}_{i_1^0,i_2^0,\ldots,i_j^0}$ denotes the n-dimensional vector of all ones except with zeros at indices $\{i_1, i_2, \ldots, i_j\}$. Parameter l is called the *masking level*, denoting the maximum number of input vector entries to be replaced with zeros.

Masking level l is also the maximum Hamming distance $d(\mathbf{x}, \mathbf{z})$ permitted to produce $(\Phi(\mathbf{x}))^T\mathbf{M}_l\Phi(\mathbf{z}) \neq 0$. Weights $\alpha_{i_1,i_2,\ldots,i_j}$ are usually expected to be some decreasing function of j, the index with respect to the masking level. The purpose of this is to give more weight towards the masking of fewer entries relative to the masking of many entries. We typically set $\alpha_{i_1,i_2,\ldots,i_j} = 2^{-wj}$ for some exponential weight $w \geq 0$. Increasing w results in larger relative weights between masking levels. For example, $w = 1$ gives weight $\frac{1}{2}$ for a masking of one entry and weight $\frac{1}{4}$ for a masking of two entries, resulting in a factor of two difference between these weights. Alternatively, $w = 2$ gives weight $\frac{1}{4}$ for a maskings of one entry and weight

$\frac{1}{16}$ for a masking of two entries, resulting in a factor of 4 difference between these weights. Without the weights, or equivalently with $w = 0$, the masking matrix has an inherent favoring of little or no entry masking as demonstrated in the next example.

Example 4. *Let* $\mathbf{x} = (\, x_1 \; x_2 \; x_3 \,)^T$ *and* $\mathbf{z} = (\, z_1 \; z_2 \; z_3 \,)^T$ *be bipolar with* $x_1 \neq z_1$, $x_2 = z_2$, *and* $x_3 = z_3$. *Let*

$$\mathbf{M}_1 = \text{diag}\left(\Phi\begin{pmatrix}1\\1\\1\end{pmatrix} + \Phi\begin{pmatrix}0\\1\\1\end{pmatrix} + \Phi\begin{pmatrix}1\\0\\1\end{pmatrix} + \Phi\begin{pmatrix}1\\1\\0\end{pmatrix} \right),$$

be a masking matrix with $l = 1$ *and* $w = 0$. *Then by Proposition 1 and Theorem 1,*

$$\begin{aligned}
&(\Phi(\mathbf{x}))^T \mathbf{M}_1 \Phi(\mathbf{x}) \\
&= \left(\Phi\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} \right)^T \left(\Phi\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} + \Phi\begin{pmatrix}0\\x_2\\x_3\end{pmatrix} + \Phi\begin{pmatrix}x_1\\0\\x_3\end{pmatrix} + \Phi\begin{pmatrix}x_1\\x_2\\0\end{pmatrix} \right) \\
&= 2^3 + 2^2 + 2^2 + 2^2 = 20,
\end{aligned}$$

and

$$\begin{aligned}
&(\Phi(\mathbf{z}))^T \mathbf{M}_1 \Phi(\mathbf{x}) \\
&= \left(\Phi\begin{pmatrix}z_1\\z_2\\z_3\end{pmatrix} \right)^T \left(\Phi\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} + \Phi\begin{pmatrix}0\\x_2\\x_3\end{pmatrix} + \Phi\begin{pmatrix}x_1\\0\\x_3\end{pmatrix} + \Phi\begin{pmatrix}x_1\\x_2\\0\end{pmatrix} \right) \\
&= 0 + 2^2 + 0 + 0 = 4.
\end{aligned}$$

If $\mathbf{x}$ *and* $\mathbf{z}$ *were learned with labels* $\mathbf{y_x}$ *and* $\mathbf{y_z}$*, respectively, then*

$$\frac{(\mathbf{y_x}(\Phi(\mathbf{x}))^T + \mathbf{y_z}(\Phi(\mathbf{z}))^T)\mathbf{M}_1\Phi(\mathbf{x})}{(\Phi(\mathbf{x}) + \Phi(\mathbf{z}))^T\mathbf{M}_1\Phi(\mathbf{x})} = \frac{20\mathbf{y_x} + 4\mathbf{y_z}}{24},$$

giving more relative weight to $\mathbf{y_x}$ *than* $\mathbf{y_z}$ *in the label prediction on target vector* $\mathbf{x}$*.*

Example 4 demonstrates that even without weights applied to the entry maskings, the masking matrix retains an inherent weighting that favors the masking of fewer input entries. However, the weights are important in amplifying this effect to better adapt the processing unit performance to the application at hand. Both exponential weight w and masking level l can be adjusted as hyperparameters.

Note that the masking matrix construction effectively involves the generation of all combinations of entry maskings up to l masked entries. This has algorithmic complexity $O(2^n \sum_{i=0}^{l} \binom{n}{i})$, or $O(4^n)$ if $l = n$, which is substantially expensive if n and l are sufficiently large. Hence, it is expected that $l \ll n$ to avoid this. The next chapter introduces an alternative masking matrix construction with more reasonable algorithmic complexity but removes the masking level as an adjustable hyperparameter.

As hinted at in Example 4, the masking matrix fulfills its role within the empirical probability formula. Finally, this establishes the generalization mechanism of the processing unit architecture.

Definition 6. Let $(\mathbf{D}, \mathbf{C})_{n,m}$ be a processing unit. The *empirical probability with respect to masking matrix* $\mathbf{M}_l$ of n-dimensional bipolar input $\mathbf{z}$, denoted $\rho(\mathbf{z}, \mathbf{M}_l)$, is

defined as

$$\rho(\mathbf{z}, \mathbf{M}_l) = \begin{cases} \frac{1}{2}\left(\frac{\mathbf{DM}_l\Phi(\mathbf{z})}{\mathbf{CM}_l\Phi(\mathbf{z})} + \mathbf{e}\right) & \text{if } \mathbf{CM}_l\Phi(\mathbf{z}) \neq 0, \\ \frac{1}{2}\mathbf{e} & \text{if } \mathbf{CM}_l\Phi(\mathbf{z}) = 0. \end{cases} \tag{2.4}$$

This concludes the background description of the THPAM processing unit architecture.

2.4 Summary

This chapter includes a succinct description of the THPAM processing unit to provide the audience with sufficient background material to understand the remaining content of this dissertation. As noted before, the background description is not exhaustive, as there are remaining components within the original publications that remain unused in this work. Refer to these for more detailed explanations and ideas for future work related to THPAM.

In summary, bipolar input and label pairs are stored within the processing unit expansion correlation matrices via a procedure akin to the Hebbian learning rule. The inputs themselves are not actually stored, but rather their representation as orthogonal expansions are stored. The storage scheme and the multiplicative properties of the orthogonal expansions are utilized to extract learned information for label predictions on target bipolar input vectors. Label predictions can be equated to an average, or weighted average with generalization, over the labels learned in association with input vectors which equate to the target vector. An additional proposed multiplicative property of orthogonal expansions is utilized to construct a

generalization mechanism by way of entry masking, hiding target vector entries in order to include the labels of similar learned inputs in the label prediction of the target vector.

Chapter 3

Theoretical Additions

3.1 Overview

This chapter specifies any additional theories pertaining to the THPAM processing unit, submitted as original contributions in this work. A more general version of the multiplicative property introduced in Proposition 1 is proved in Section 3.2, which has several applications. The property can be used to construct particular masking matrices in an alternative manner with somewhat more reasonable algorithmic complexity, but at the expense of removing the masking level as an adjustable hyperparameter. This alternative masking matrix construction is presented in Section 3.3. In Section 3.4, an alternative generalization mechanism is proposed in which target bipolar vector entries are flipped via orthogonal expansions instead of masked in order to include the learned labels of similar vectors in the empirical probability. Section 3.5 demonstrates a unique application for the entry flipping mechanism in the update rule of the expansion correlation matrices to learn data clusters within a specified Hamming distance from an input vector in a single learning rendition rather than an amount of executions proportional to the number of vectors belonging to the data cluster. Section 3.6 shows that the entry flipping mechanism can also be used to equate the empirical probability formula to a weighted average over the training set without the use of the orthogonal expansion.

This can be used to simulate the performance of the THPAM processing unit on data sets that are too large in input size to be reasonably encoded by the orthogonal expansion with modern computer architectures.

3.2 Multiplication of Orthogonal Expansions

The property discussed herein is a more general version of Proposition 1 introduced in the previous chapter. Proposition 1 stated that the multiplicative act of replacing elements in a vector with zeros is preserved over the orthogonal expansion, meaning entry masking can occur via entrywise multiplication between orthogonal expansions rather than between the original vectors. This is important because the THPAM processing unit stores and operates on orthogonal expansions rather than the original vectors. Entry masking enables the THPAM processing unit to generalize when producing label predictions, permitting similar learned vectors to contribute nontrivially towards the label prediction of a target input. The proposed theorem demonstrates that the property invoked in entry masking also exists between any two ternary vectors rather than only between a binary vector and a bipolar vector.

Theorem 3. *Let* $\mathbf{a} = \begin{pmatrix} a_1 & a_2 & \cdots & a_n \end{pmatrix}^T$ *and* $\mathbf{b} = \begin{pmatrix} b_1 & b_2 & \cdots & b_n \end{pmatrix}^T$ *be ternary vectors. Then,*

$$\Phi(\mathbf{a} \circ \mathbf{b}) = \Phi(\mathbf{a}) \circ \Phi(\mathbf{b}),$$

where $\mathbf{a} \circ \mathbf{b}$ *is the entrywise product operation between vectors* $\mathbf{a}$ *and* $\mathbf{b}$*.*

Proof. A proof by induction can be used to demonstrate that

$$\Phi_i(\mathbf{a} \circ \mathbf{b}) = \Phi_i(\mathbf{a}) \circ \Phi_i(\mathbf{b})$$

for all $i \in \{1, 2, \ldots, n\}$.

(Base Step): Clearly, $\Phi_1(\mathbf{a} \circ \mathbf{b}) = \Phi_1(\mathbf{a}) \circ \Phi_1(\mathbf{b}) = \begin{pmatrix} 1 & a_1 b_1 \end{pmatrix}^T$.

(Inductive Step): Suppose that for $i \in \{1, 2, \ldots, n-1\}$,

$$\Phi_i(\mathbf{a} \circ \mathbf{b}) = \Phi_i(\mathbf{a}) \circ \Phi_i(\mathbf{b}).$$

Then by Definition 2,

$$\begin{aligned}
\Phi_{i+1}(\mathbf{a} \circ \mathbf{b}) &= \begin{pmatrix} \Phi_i(\mathbf{a} \circ \mathbf{b}) \\ a_{i+1} b_{i+1} \Phi_i(\mathbf{a} \circ \mathbf{b}) \end{pmatrix} = \begin{pmatrix} \Phi_i(\mathbf{a}) \circ \Phi_i(\mathbf{b}) \\ a_{i+1} b_{i+1} \Phi_i(\mathbf{a}) \circ \Phi_i(\mathbf{b}) \end{pmatrix} \\
&= \begin{pmatrix} \Phi_i(\mathbf{a}) \\ a_{i+1} \Phi_i(\mathbf{a}) \end{pmatrix} \circ \begin{pmatrix} \Phi_i(\mathbf{b}) \\ b_{i+1} \Phi_i(\mathbf{b}) \end{pmatrix} \\
&= \Phi_{i+1}(\mathbf{a}) \circ \Phi_{i+1}(\mathbf{b}).
\end{aligned}$$

This completes the proof. □

Theorem 3 demonstrates that the orthogonal expansion of an entrywise product between two ternary vectors is equivalent to the entrywise product of their respective orthogonal expansions. This verifies that entry masking via orthogonal expansion is equivalent to masking entries in the original vector and subsequently computing its orthogonal expansion. However, this result is more general in that this underlying relationship is not exclusive to the process of entry masking but also applicable between any pair of ternary vectors. The remaining contents of this

chapter make extensive use of Theorem 3 in devising several tools relating to the processing unit architecture.

3.3 Alternative Masking Matrix Construction

Theorem 3 can be used to devise an alternative method of constructing masking matrices. To introduce this approach, we first visit an example pertaining to the masking of multiple entries of a bipolar vector via orthogonal expansions.

Example 5. *Let* $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$ *be a bipolar vector, and suppose we wish to mask to its last two entries via its orthogonal expansion. The usual way to approach this would be to perform the product*

$$\operatorname{diag}\left(\Phi\begin{pmatrix}1\\0\\0\end{pmatrix}\right)\Phi\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} = \Phi\begin{pmatrix}x_1\\0\\0\end{pmatrix},$$

but an alternative way would be to apply two single entry masks in succession, as in the product

$$\operatorname{diag}\left(\Phi\begin{pmatrix}1\\0\\1\end{pmatrix}\right)\operatorname{diag}\left(\Phi\begin{pmatrix}1\\1\\0\end{pmatrix}\right)\Phi\begin{pmatrix}x_1\\x_2\\x_3\end{pmatrix} = \operatorname{diag}\left(\Phi\begin{pmatrix}1\\0\\1\end{pmatrix}\right)\Phi\begin{pmatrix}x_1\\x_2\\0\end{pmatrix}$$

$$= \Phi\begin{pmatrix}x_1\\0\\0\end{pmatrix}.$$

Computationally, this doubles the resources required to produce the desired result, but there is an additional interpretation which leads to less computational expense. According to the general definition of the masking matrix in Definition 5, the three maskings $\Phi\begin{pmatrix}1\\0\\1\end{pmatrix}$, $\Phi\begin{pmatrix}1\\1\\0\end{pmatrix}$, *and* $\Phi\begin{pmatrix}1\\0\\0\end{pmatrix}$ *would all be produced individually. Alternatively,* $\Phi\begin{pmatrix}1\\0\\1\end{pmatrix}$ *and* $\Phi\begin{pmatrix}1\\1\\0\end{pmatrix}$ *could be individually generated, and their entrywise product can be used to generate* $\Phi\begin{pmatrix}1\\0\\0\end{pmatrix}$, *as in*

$$\Phi\begin{pmatrix}1\\0\\1\end{pmatrix} \circ \Phi\begin{pmatrix}1\\1\\0\end{pmatrix} = \Phi\begin{pmatrix}1\\0\\0\end{pmatrix}.$$

The algorithmic complexity of masking matrix construction remains unaltered in Example 5 due to the low masking level that was considered. However, should one desire a masking matrix consisting of all possible combinations of entry masks, this property must be utilized to reduce the computational expense. The following corollary summarizes the procedure of multiplying two entry maskings to create a new entry masking.

Corollary 1. *Let* $\mathbf{e}_{i_1^0,i_2^0,\ldots,i_r^0}$ *and* $\mathbf{e}_{j_1^0,j_2^0,\ldots,j_s^0}$ *be the n-dimensional vectors of all ones, except with zeros at indices* $\{i_1, i_2, \ldots, i_r\}$ *and* $\{j_1, j_2, \ldots, j_s\}$, *respectively. Let* $\{k_1, k_2, \ldots, k_q\} = \{i_1, i_2, \ldots, i_r\} \cup \{j_1, j_2, \ldots, j_s\}$. *Then*

$$\Phi(\mathbf{e}_{i_1^0,i_2^0,\ldots,i_r^0}) \circ \Phi(\mathbf{e}_{j_1^0,j_2^0,\ldots,j_s^0}) = \Phi(\mathbf{e}_{k_1^0,k_2^0,\ldots,k_q^0})$$

Proof. By Theorem 3,

$$\begin{aligned}\Phi(\mathbf{e}_{i_1^0,i_2^0,\ldots,i_r^0}) \circ \Phi(\mathbf{e}_{j_1^0,j_2^0,\ldots,j_s^0}) &= \Phi(\mathbf{e}_{i_1^0,i_2^0,\ldots,i_r^0} \circ \mathbf{e}_{j_1^0,j_2^0,\ldots,j_s^0}) \\ &= \Phi(\mathbf{e}_{k_1^0,k_2^0,\ldots,k_q^0})\end{aligned}$$

This completes the proof. □

Corollary 1 demonstrates more extensively that the product of two masking matrices produces yet another masking matrix which includes all entry maskings contained in the original matrices in addition to new entry maskings. The presence of the identity masking $\Phi(\mathbf{e})$ in every masking matrix plays a significant role in this. For example, if masking matrices $\text{diag}(\Phi(\mathbf{e}) + \Phi(\mathbf{e}_{1^0}))$ and $\text{diag}(\Phi(\mathbf{e}) + \Phi(\mathbf{e}_{2^0}))$ are given, then their product

$$\text{diag}(\Phi(\mathbf{e}) + \Phi(\mathbf{e}_{1^0}))\text{diag}(\Phi(\mathbf{e}) + \Phi(\mathbf{e}_{2^0})) = \text{diag}(\Phi(\mathbf{e}) + \Phi(\mathbf{e}_{1^0}) + \Phi(\mathbf{e}_{2^0}) + \Phi(\mathbf{e}_{1^0,2^0})),$$

is yet another masking matrix containing the original maskings $\Phi(\mathbf{1}_{1^0})$ and $\Phi(\mathbf{1}_{2^0})$ as well as their combination $\Phi(\mathbf{1}_{1^0,2^0})$. This property can be used to more efficiently generate a masking matrix with masking level l equal to the input dimension n in the following manner:

$$\mathbf{M}_n = \prod_{i=1}^{n} \text{diag}\left(\Phi(\mathbf{e}) + \alpha_i \Phi(\mathbf{e}_{i^0})\right), \tag{3.1}$$

where $\alpha_i > 0$ are scalar weights, and $\mathbf{e}$ and $\mathbf{e}_{i^0}$ are defined as before. Typically, $\alpha_i = 2^{-w}$ for all i, where parameter w is an adjustable exponential weight. The algorithmic complexity of generating this masking matrix is $O(n2^n)$, which remains intractable but is preferable to the $O(4^n)$ complexity involved in following the original definition to generate a masking matrix with masking level $l = n$. However,

there are some disadvantages to this alternative construction. There is less freedom in choosing the weights associated with the different combinations of maskings in the underlying formula, since any combination of masks would have a weighting strictly proportional to the product of the involved individual entry masks. For example, if the first three entries in a vector are being masked, then the weight associated with this mask must be $\alpha_1\alpha_2\alpha_3$ rather than a custom weight $\alpha_{1,2,3}$ afforded in the original masking matrix formula. Furthermore, this is strictly a masking matrix with masking level $l = n$ and cannot be otherwise, so it lacks the freedom of adjusting the masking level as a hyperparameter. This is usually of little concern providing w is sufficiently large, as experimentally demonstrated in Chapter 5.

3.4 Alternative Generalization with Entry Flipping

Theorem 3 can also be used to devise an alternative generalization method to entry masking. Instead of masking differing vector entries in order to relate a target input to similar learned vectors, these entries could be flipped like a switch as is an inherent mechanism of bipolar and binary variables. The objective of entry masking was to produce a nonzero inner product of the orthogonal expansions of two similar vectors by replacing their differing entries with zeros, thereby removing those entries from the inner product and relaxing the orthogonality between these expansions. A comparable result can be achieved by extracting each pair of differing entries and flipping the value of one entry in each pair. For example, if $\mathbf{x}$ and $\mathbf{z}$ are bipolar and equal except for a distortion at their kth entries, i.e. $x_k \neq z_k$, then

they can be made equivalent by replacing z_k with $-z_k$. To introduce this approach, we first visit an example in which entry flipping is used to produce a nonzero inner product between the orthogonal expansions of two different bipolar vectors.

Example 6. *Let* $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$ *and* $\mathbf{z} = \begin{pmatrix} z_1 & z_2 & z_3 \end{pmatrix}^T$ *be bipolar, with* $x_1 \neq z_1$, $x_2 = z_2$, *and* $x_3 = z_3$. *Then* $(\Phi(\mathbf{x}))^T\Phi(\mathbf{z}) = 0$, *and we wish to alter this inner product to produce a nonzero value by flipping the value of* z_1 *via orthogonal expansions. Using the original vectors, this can be accomplished by performing the product*

$$\text{diag}\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} -z_1 \\ z_2 \\ z_3 \end{pmatrix} = \mathbf{x}.$$

By Theorem 3,

$$\text{diag}\left(\Phi\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}\right)\Phi\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \Phi\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} \circ \Phi\begin{pmatrix} z_1 \\ z_2 \\ z_3 \end{pmatrix} = \Phi\begin{pmatrix} -z_1 \\ z_2 \\ z_3 \end{pmatrix} = \Phi(\mathbf{x}),$$

and by Theorem 1,

$$(\Phi(\mathbf{x}))^T\text{diag}\left(\Phi\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix}\right)\Phi(\mathbf{z}) = (\Phi(\mathbf{x}))^T\Phi(\mathbf{x}) = 2^3 = 8,$$

which accomplishes the objective via orthogonal expansions.

Notice that the entry flipping procedure is closely related to that of entry masking. In fact, this example is nearly the same as Example 2 exemplifying entry

masking. The notable difference is that entry maskings use zeros to hide corresponding entries in a target vector, whereas the entry flipping matrix uses negatives to flip the corresponding entries in a target vector from positive to negative or vice versa. A matrix similar to a masking matrix can thereby be used as an alternative generalization mechanism to entry masking.

Definition 7 (Flipping Matrix)**.** A matrix $\mathbf{B}_l$ is a *flipping matrix* if it is of the form

$$\mathbf{B}_l = \operatorname{diag}\left(\Phi(\mathbf{e}) + \sum_{j=1}^{l}\sum_{i_j=j}^{n}\cdots\sum_{i_2=2}^{i_3-1}\sum_{i_1=1}^{i_2-1}\alpha_{i_1,i_2,\ldots,i_j}\Phi(\mathbf{e}_{i_{1-},i_{2-},\ldots,i_{j-}})\right),$$

where all $\alpha_{i_1,i_2,\ldots,i_j} > 0$ are scalar weights, $\mathbf{e}$ denotes the n-dimensional vector of all ones, and $\mathbf{e}_{i_{1-},i_{2-},\ldots,i_{j-}}$ denotes the n-dimensional vector of all ones except with negatives at indices $\{i_1, i_2, \ldots, i_j\}$. Parameter l denotes the maximum number of input vector entries to be flipped.

Typically, we set $\alpha_{i_1,i_2,\ldots,i_j} = 2^{-wj}$ where parameter w is an adjustable exponential weight. Like with the masking matrix, l is also the maximum Hamming distance $d(\mathbf{x}, \mathbf{z})$ permitted to produce $(\Phi(\mathbf{x}))^T\mathbf{B}_l\Phi(\mathbf{z}) \neq 0$. The flipping matrix in Definition 7 follows the same structure of the masking matrix in Definition 5, except for the use of negatives in place of zeros. Thus, constructing the flipping matrix in this manner has the same algorithmic complexity of $O(2^n\sum_{i=0}^{l}\binom{n}{i})$ and $O(4^n)$ when $l = n$. However, just as with the masking matrix, there is a construction method with $O(n2^n)$ complexity to produce a flipping matrix with $l = n$, thanks to Theorem 3:

$$\mathbf{B}_n = \prod_{i=1}^{n}\operatorname{diag}\left(\Phi(\mathbf{e}) + \alpha_i\Phi(\mathbf{e}_{i^-})\right), \tag{3.2}$$

where $\alpha_i > 0$ are scalar weights, and $\mathbf{e}$ and $\mathbf{e}_{i-}$ are defined as before. Like the alternative masking matrix construction, this manner of construction restricts the freedom of the individual selection of weights for each combination of entry flippings. It also removes the freedom to adjust l as a hyperparameter. Typically, we set $\alpha_i = 2^{-w}$ for all i, where parameter w is an adjustable exponential weight.

There is a notable difference between the bilinear forms $(\Phi(\mathbf{x}))^T\mathbf{M}_l\Phi(\mathbf{z})$ and $(\Phi(\mathbf{x}))^T\mathbf{B}_l\Phi(\mathbf{z})$. We previously observed in Example 4 that without any entry masking weights, i.e. $w = 0$, the masking matrix retains an inherent weighting mechanism favoring fewer masked entries. This is because there are often many ways to perform entry masking on $\Phi(\mathbf{z})$ to achieve a nonzero inner product with $\Phi(\mathbf{x})$. All of the entries in $\mathbf{z}$ that differ from those in $\mathbf{x}$ would necessarily have to be masked, but any of the remaining entries in $\mathbf{z}$ could also be masked in any combination to contribute nontrivially to $(\Phi(\mathbf{x}))^T\mathbf{M}_l\Phi(\mathbf{z})$, providing $l > d(\mathbf{x}, \mathbf{z})$. Conversely, the entry flipping matrix contains at most one entry flipping which contributes nontrivially to the value of $(\Phi(\mathbf{x}))^T\mathbf{B}_l\Phi(\mathbf{z})$. All of the entries in $\mathbf{z}$ which differ from those in $\mathbf{x}$ must be flipped to avoid orthogonality, but the flipping of any remaining entries will reintroduce orthogonality. This behavior is summarized in the below lemma.

Lemma 1. *Let* $\mathbf{x}$ *and* $\mathbf{z}$ *be* n*-dimensional bipolar vectors. Let* $\mathbf{B}_l$ *be a general flipping matrix with* $l \geq d(\mathbf{x}, \mathbf{z})$ *in Hamming distance. Then*

$$(\Phi(\mathbf{x}))^T\mathbf{B}_l\Phi(\mathbf{z}) = \begin{cases} 2^n & \text{if } \mathbf{x} = \mathbf{z}, \\ \alpha_{i_1,i_2,\ldots,i_k}2^n & \text{if } \mathbf{x} \neq \mathbf{z}, \end{cases}$$

where $k = d(\mathbf{x}, \mathbf{z})$.

Proof. If $\mathbf{x} \neq \mathbf{z}$, then there are entries $x_{i_1}, x_{i_2}, \ldots x_{i_k}$ in $\mathbf{x}$ and entries $z_{i_1}, z_{i_2}, \ldots z_{i_k}$ in $\mathbf{z}$ such that $x_{i_j} \neq z_{i_j}$ for $j = 1, 2, \ldots, k$, where $k = d(\mathbf{x}, \mathbf{z})$. Since $\mathbf{x}$ and $\mathbf{z}$ are bipolar, $x_{i_j} = -z_{i_j}$ for $j = 1, 2, \ldots, k$. Then there is a unique bipolar vector $\mathbf{e}_{i_1^-, i_2^-, \ldots, i_k^-}$ such that

$$\mathbf{x} = \mathbf{e}_{i_1^-, i_2^-, \ldots, i_k^-} \circ \mathbf{z}.$$

By Theorem 3,

$$\Phi(\mathbf{x}) = \Phi(\mathbf{e}_{i_1^-, i_2^-, \ldots, i_k^-}) \circ \Phi(\mathbf{z}) = \mathrm{diag}(\Phi(\mathbf{e}_{i_1^-, i_2^-, \ldots, i_k^-}))\Phi(\mathbf{z}),$$

where $\Phi(\mathbf{1}_{i_1^-, i_2^-, \ldots, i_k^-})$ is included in the linear combination along the diagonal of $\mathbf{B}_l$ with weight $\alpha_{i_1, i_2, \ldots, i_k}$. Then by Theorem 1,

$$\begin{aligned}
(\Phi(\mathbf{x}))^T \mathbf{B}_l \Phi(\mathbf{z}) &= (\Phi(\mathbf{x}))^T \mathrm{diag}(\Phi(\mathbf{e}) + \alpha_1 \Phi(\mathbf{e}_{1^-}) + \alpha_2 \Phi(\mathbf{e}_{2^-}) + \ldots \\
&\quad + \alpha_{i_1, i_2, \ldots, i_k} \Phi(\mathbf{e}_{i_1^-, i_2^-, \ldots, i_k^-}) + \ldots + \alpha_{i_1, i_2, \ldots, i_l} \Phi(\mathbf{e}_{i_1^-, i_2^-, \ldots, i_l^-}))\Phi(\mathbf{z}) \\
&= (\Phi(\mathbf{x}))^T \mathrm{diag}(\Phi(\mathbf{e}))\Phi(\mathbf{z}) + \alpha_1 (\Phi(\mathbf{x}))^T \mathrm{diag}(\Phi(\mathbf{e}_{1^-}))\Phi(\mathbf{z}) + \ldots \\
&\quad + \alpha_{i_1, i_2, \ldots, i_k} (\Phi(\mathbf{x}))^T \mathrm{diag}(\Phi(\mathbf{e}_{i_1^-, i_2^-, \ldots, i_k^-}))\Phi(\mathbf{z}) + \ldots \\
&\quad + \alpha_{i_1, i_2, \ldots, i_l} (\Phi(\mathbf{x}))^T \mathrm{diag}(\Phi(\mathbf{e}_{i_1^-, i_2^-, \ldots, i_l^-}))\Phi(\mathbf{z}) \\
&= 0 + \ldots + \alpha_{i_1, i_2, \ldots, i_k} (\Phi(\mathbf{x}))^T \mathrm{diag}(\Phi(\mathbf{e}_{i_1^-, i_2^-, \ldots, i_k^-}))\Phi(\mathbf{z}) + 0 + \ldots + 0 \\
&= \alpha_{i_1, i_2, \ldots, i_k} (\Phi(\mathbf{x}))^T \Phi(\mathbf{x}) \\
&= \alpha_{i_1, i_2, \ldots, i_k} 2^n
\end{aligned}$$

Alternatively, if $\mathbf{x} = \mathbf{z}$ then

$$(\Phi(\mathbf{x}))^T \mathbf{B}_l \Phi(\mathbf{z}) = (\Phi(\mathbf{x}))^T \Phi(\mathbf{x}) = 2^n.$$

This completes the proof. □

Lemma 1 establishes that the exact value of $(\Phi(\mathbf{x}))^T \mathbf{B}_l \Phi(\mathbf{z})$ with $l \geq d(\mathbf{x}, \mathbf{z})$ is always proportional only to 2^n and the weight associated with the unique entry flipping needed to produce $\Phi(\mathbf{x})$ from $\Phi(\mathbf{z})$. This also demonstrates the important distinction between the masking and flipping matrices, namely that several combinations of entry maskings in $\mathbf{M}_l$ can contribute nontrivially to $(\Phi(\mathbf{x}))^T \mathbf{M}_l \Phi(\mathbf{z})$ whereas at most one entry flipping in $\mathbf{B}_l$ contributes nontrivially to $(\Phi(\mathbf{x}))^T \mathbf{B}_l \Phi(\mathbf{z})$. This distinction provides some unique use cases for entry flipping, which are introduced ahead.

3.5 Efficient Learning of Data Clusters

The entry flipping matrix presents a unique application towards the learning phase of the processing unit architecture. To introduce this application, we first visit an example pertaining to the product of an entry flipping matrix with an orthogonal expansion of a bipolar vector.

Example 7. *Let* $\mathbf{B}_1$ *be the entry flipping matrix for input dimension* $n = 3$ *with masking level* $l = 1$ *and exponential weight* $w = 0$*, specifically,*

$$\mathbf{B}_1 = \text{diag}\left(\Phi\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} + \Phi\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} + \Phi\begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} + \Phi\begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} \right).$$

Let $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$ *be a bipolar input vector. Then by invoking Theorem 3,*

$$\mathbf{B}_1\Phi(\mathbf{x}) = \text{diag}\left(\Phi\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} + \Phi\begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} + \Phi\begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} + \Phi\begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} \right) \Phi\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

$$= \Phi\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \Phi\begin{pmatrix} -x_1 \\ x_2 \\ x_3 \end{pmatrix} + \Phi\begin{pmatrix} x_1 \\ -x_2 \\ x_3 \end{pmatrix} + \Phi\begin{pmatrix} x_1 \\ x_2 \\ -x_3 \end{pmatrix},$$

which is the sum of the orthogonal expansions of $\mathbf{x}$ *and each vector within a unit of Hamming distance away from* $\mathbf{x}$*. If the masking level were* $l = 2$*, then the same terms above would be produced along with the orthogonal expansions of the vectors at two Hamming distance units away from* $\mathbf{x}$*. Suppose that* $\mathbf{x}$ *and the vectors within unit Hamming distance from* $\mathbf{x}$ *must be learned with the same bipolar label* $\mathbf{y}$*. Normally, this would require four separate learning renditions to update processing unit* $(\mathbf{D}, \mathbf{C})$ *with*

$$\mathbf{D} \leftarrow \mathbf{D} + \mathbf{y}\left(\Phi\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \right)^T + \mathbf{y}\left(\Phi\begin{pmatrix} -x_1 \\ x_2 \\ x_3 \end{pmatrix} \right)^T$$

$$+ \mathbf{y}\left(\Phi\begin{pmatrix} x_1 \\ -x_2 \\ x_3 \end{pmatrix} \right)^T + \mathbf{y}\left(\Phi\begin{pmatrix} x_1 \\ x_2 \\ -x_3 \end{pmatrix} \right)^T,$$

and

$$\mathbf{C} \leftarrow \mathbf{C} + \left(\Phi \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \Phi \begin{pmatrix} -x_1 \\ x_2 \\ x_3 \end{pmatrix} + \Phi \begin{pmatrix} x_1 \\ -x_2 \\ x_3 \end{pmatrix} + \Phi \begin{pmatrix} x_1 \\ x_2 \\ -x_3 \end{pmatrix} \right)^T,$$

but the same result is produced by performing

$$\mathbf{D} \leftarrow \mathbf{D} + \mathbf{y}(\mathbf{B}_1\Phi(\mathbf{x}))^T,$$

$$\mathbf{C} \leftarrow \mathbf{C} + (\mathbf{B}_1\Phi(\mathbf{x}))^T,$$

in a single learning phase.

The procedure in Example 7 can be extended to any input size and any masking level. In general, if the expansion correlation matrices are to be updated in processing unit $(\mathbf{D}, \mathbf{C})_{n,m}$ on bipolar pair $(\mathbf{x}, \mathbf{y})$ along with vectors within l units of Hamming distance from $\mathbf{x}$, then this can be accomplished by updating the processing unit with

$$\mathbf{D} \leftarrow \mathbf{D} + \mathbf{y}(\mathbf{B}_l\Phi(\mathbf{x}))^T,$$

$$\mathbf{C} \leftarrow \mathbf{C} + (\mathbf{B}_l\Phi(\mathbf{x}))^T.$$

This has algorithmic complexity $O(m2^n)$, which is a preferable alternative to the $O(m2^n \sum_{i=0}^{l} \binom{n}{i})$ complexity of performing separate learning renditions.

3.6 Simulating the Processing Unit on Large Inputs

The $O(2^n)$ complexity of the orthogonal expansion imposes a severe limitation on the input size n to be reasonably processed on common computer architectures.

The anticipated yearly improvements to processing power and memory access on traditional architectures are insufficient in satisfying the resource demand of exponentially complex algorithms in a meaningful way. Future architectures could prove promising for THPAM if there is a compatible processing unit implementation. The advent of quantum computing boasts linear running time on some exponentially complex problems, which could substantially reduce the run time of a compatible processing unit program. Quantum computing is currently immature in its development, so this option remains unavailable at this time. For now, Lemma 1 provides one method of circumventing the orthogonal expansion by equating the empirical probability formula to a weighted average over the learned output vectors, as proposed in the next corollary.

Corollary 2. *Let* $(\mathbf{D}, \mathbf{C})_{n,m}$ *be a processing unit trained on bipolar pairs* $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_N, \mathbf{y}_N)\}$*, and let* $\mathbf{z}$ *be an* n*-dimensional bipolar vector. Let* $\mathbf{B}_n$ *be the entry flipping matrix*

$$\mathbf{B}_n = \prod_{i=1}^{n} \operatorname{diag}\left(\Phi(\mathbf{e}) + \alpha\Phi(\mathbf{e}_{i^-})\right),$$

where $\alpha > 0$*. Then*

$$\frac{\mathbf{D}\mathbf{B}_n\Phi(\mathbf{z})}{\mathbf{C}\mathbf{B}_n\Phi(\mathbf{z})} = \frac{\sum_{i=1}^{N} \alpha^{d(\mathbf{x}_i,\mathbf{z})}\mathbf{y}_i}{\sum_{i=1}^{N} \alpha^{d(\mathbf{x}_i,\mathbf{z})}}, \tag{3.3}$$

where $d(\cdot,\cdot)$ *is the Hamming distance metric.*

Proof. By Lemma 1,

$$\begin{aligned}\frac{\mathbf{DB}_n\Phi(\mathbf{z})}{\mathbf{CB}_n\Phi(\mathbf{z})} &= \frac{\sum_{i=1}^{N}\mathbf{y}_i(\Phi(\mathbf{x}_i))^T\mathbf{B}_n\Phi(\mathbf{z})}{\sum_{i=1}^{N}(\Phi(\mathbf{x}_i))^T\mathbf{B}_n\Phi(\mathbf{z})} \\ &= \frac{\sum_{i=1}^{N}2^n\alpha^{d(\mathbf{x}_i,\mathbf{z})}\mathbf{y}_i}{\sum_{i=1}^{N}2^n\alpha^{d(\mathbf{x}_i,\mathbf{z})}} \\ &= \frac{\sum_{i=1}^{N}\alpha^{d(\mathbf{x}_i,\mathbf{z})}\mathbf{y}_i}{\sum_{i=1}^{N}\alpha^{d(\mathbf{x}_i,\mathbf{z})}}\end{aligned}$$

This completes the proof. □

Corollary 2 provides a manner for simulating the performance of a THPAM processing unit without the use of orthogonal expansions. The empirical probability formula is reduced to a weighted average over the training labels, with the weights determined by the Hamming distance between the learned inputs and the target vector. This is useful when applying the processing unit to data sets with input size large enough to cause the orthogonal expansion to be intractable. However, this cannot be used for online learning since this weighted average requires storage of the entire data set in order to compute the hamming distances between a target vector and each of the trained vectors. Additional training inputs require proportionally more storage space and processing time in generating predictions on target vectors. The benefit of fixed storage size granted by the processing unit expansion correlation matrices is lost. Nevertheless, Corollary 2 is a useful result for purposes of simulation and analysis.

3.7 Summary

This concludes the presentation of original theoretical findings relating to the THPAM processing unit architecture. All of these contributions rely on Theorem 3 in varying extent, which demonstrates that the entrywise product between ternary vectors is preserved under the orthogonal expansion. This result has utility in devising an alternative masking matrix construction and the alternative generalization mechanism of entry flipping. Entry flipping can be uniquely applied in the processing unit learning phase in order to efficiently learn data clusters within a specified Hamming distance from a central bipolar input. Entry flipping can also be used to reduce the empirical probability formula to a weighted average that does not contain orthogonal expansions, which cannot be used for online learning but can be used for experimental purposes.

Chapter 4

Programming Implementation

4.1 Overview

This chapter describes the programming implementation and computer hardware utilized in demonstrating and examining the performance of the THPAM processing unit on sample datasets. The processing unit is implemented as a parallel C program with MPI architecture suitable for communication among multiple parallel processes. This approach is relatively straightforward for some components of the model which mostly involve matrix algebra, which is readily parallelizable with introductory knowledge in parallel computing [18]. However, the parallelization of some core functionality is less intuitive and requires careful explanation, particularly the orthogonal expansion implementation. Although the orthogonal expansion encoding of n-dimensional ternary vectors into 2^n-dimensional ternary vectors imposes a severe bound on the acceptable size of the input vectors subject to the orthogonal expansion due to computer memory and processing limitations, the use of parallel computing remains invaluable in pushing against these limitations and permitting somewhat larger input vector dimensions than possible with a serial program.

Section 4.2 carefully describes the orthogonal expansion implementation, which is integral to the other processing unit mechanisms. Section 4.3 briefly describes the correlation learning implementation. Section 4.4 thoroughly details two programs

for masking matrix construction, the first following the general masking matrix definition and the second following the alternative approach introduced in Section 3.3. Finally, Section 4.5 describes the implementation for constructing the empirical probability on a target input. A performance study examining the scalability of the parallel program is summarized in Section 4.6. Section 4.7 examines the capability of the processing unit as a pattern recognizer on a small handcrafted example involving the ten numerical digits.

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `hpcf.umbc.edu` for more information on HPCF and the projects using its resources.

4.2 Orthogonal Expansion

The THPAM processing unit model is fundamentally constructed based on the interactions between orthogonal expansions detailed in Theorem 1. All information is stored and retrieved according to these interactions, so the orthogonal expansion implementation is to be described first. Note that the original description in Definition 2 provides an obvious recursive method of constructing the orthogonal expansion. The following is a C programming function of this recursive method

expressed iteratively.

```
void orthogonalExpansion(
        int inputSize,
        int input[],
        int expansion[])
{
    int inputIndex;
    int expansionIndex;
    int shiftSize = 1;

    expansion[0] = 1;
    for (inputIndex = 0; inputIndex < inputSize; inputIndex++) {
        for (expansionIndex = shiftSize; expansionIndex < (2 * shiftSize);
                expansionIndex++) {
            expansion[expansionIndex] =
                    input[inputIndex] * expansion[expansionIndex - shiftSize];
        }
        shiftSize *= 2;
    }
}
```

The first entry in the expansion is always 1, and the subsequent elements are produced by looping through each input vector entry and performing a product between that entry and all of the previously stored entries in the orthogonal expansion, as seen within the innermost `for` loop within the code. In order to parallelize this method it must be altered so that the single task of producing the orthogonal expansion can be split into multiple tasks which can be computed independently. Presented as is, it is unclear that such a division of labor can be achieved considering the construction of an orthogonal expansion is inherently recursive even when expressed in this iterative manner. It is true that each subsequent entry in the orthogonal expansion is dependent on all of the preceding entries, but this can still be divided into independent tasks with some expense of repetitive computation and some creative liberty in the orthogonal expansion construction.

Example 8. *Let* $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T$. *Then the orthogonal expansion of* $\mathbf{x}$ *is*

$$\Phi(\mathbf{x}) = \begin{pmatrix} 1 & x_1 & x_2 & x_1x_2 & x_3 & x_1x_3 & x_2x_3 & x_1x_2x_3 \end{pmatrix}^T.$$

We wish to divide as much of the construction of $\Phi(\mathbf{x})$ *into two equal and independent tasks. Let us assume that a small portion of this procedure has been completed to produce* $\Phi_1(\mathbf{x}) = \begin{pmatrix} 1 & x_1 \end{pmatrix}^T$ *so that two entries of the orthogonal expansion are now available. Split these two entries and continue the expansion procedure as follows:*

$$\begin{pmatrix} 1 \end{pmatrix}^T \longrightarrow \begin{pmatrix} 1 & x_2 \end{pmatrix}^T \longrightarrow \begin{pmatrix} 1 & x_2 & x_3 & x_2x_3 \end{pmatrix}^T,$$
$$\begin{pmatrix} x_1 \end{pmatrix}^T \longrightarrow \begin{pmatrix} x_1 & x_1x_2 \end{pmatrix}^T \longrightarrow \begin{pmatrix} x_1 & x_1x_2 & x_1x_3 & x_1x_2x_3 \end{pmatrix}^T.$$

Concatenate these vectors to produce

$$\begin{pmatrix} 1 & x_2 & x_3 & x_2x_3 & x_1 & x_1x_2 & x_1x_3 & x_1x_2x_3 \end{pmatrix}^T,$$

which is nothing more than a permutation of $\Phi(\mathbf{x})$*. Note that the subtasks of constructing the two halves of the concatenated vector could be completed independently of each other providing the initial two entries of the orthogonal expansion are available.*

The fact that the procedure in Example 8 produces a permutation of the orthogonal expansion rather than the literal orthogonal expansion has consequences which are minor but should be observed. Since orthogonal expansions interact with each other only via scalar addition and inner products within the processing unit architecture, the order of the expansion entries is inconsequential providing the order remains consistent among all the permuted expansions. This can be easily achieved with parallel processing by always starting with a blank processing unit when altering the number of computer processes being used. Otherwise if a processing unit in a saved state is being migrated to a new parallel machine with

a different number of computer processes, the columns of its expansion matrices must be appropriately permuted to correctly interact with any newly constructed orthogonal expansions. It is insufficient to simply concatenate and store all the local portions of the expansion correlation matrices in the manner exemplified with the orthogonal expansion in Example 8.

Below is a parallel implementation of the orthogonal expansion in which each parallel process is responsible for maintaining its own local orthogonal expansion sub-vector, denoted `l_expansion`, of dimension $\frac{2^n}{p}$ where n is the input size and p is the number of parallel processes. These sub-vectors are never concatenated into a single expansion within the program since this is an unnecessary cost. Instead, each computer processor is also responsible for its own local expansion correlation sub-matrices whose entries correspond with those of their local orthogonal expansion sub-vectors. The next section discusses this in greater detail.

```
void parallelOrthogonalExpansion(
        int inputSize,
        int input[],
        int l_expansion[],
        int processId,
        int numProcesses)
{
    int inputIndex;
    int expansionIndex;
    int shiftSize = 1;
    int exponent = (int)log2((double)numProcesses);
    int p_expansion[numProcesses];

    orthogonalExpansion(exponent, input, p_expansion);

    l_expansion[0] = p_expansion[processId];
    for (inputIndex = exponent; inputIndex < inputSize; inputIndex++) {
        for (expansionIndex = shiftSize; expansionIndex < (2 * shiftSize);
                expansionIndex++) {
            l_expansion[expansionIndex] =
                    input[inputIndex] * l_expansion[expansionIndex - shiftSize];
        }
        shiftSize *= 2;
    }
}
```

As is common in parallel programming, the parallelized orthogonal expansion

is very similar to its serial counterpart. Customarily, any variables which denote a local partition of some greater quantity are concatenated with an `l_` in their names, as seen in `l_expansion`. The variable `p_expansion` is similarly branded with a `p_` to denote a partially completed orthogonal expansion which serves as a small foundation on which to produce the remaining pieces of the expansion independently. It is necessary that the number of parallel processes used to run this program be a power of two, as in 1, 2, 4, 8, 16, and so on. This is required because partial orthogonal expansion `p_expansion` always contains a number of entries equivalent to a power of two, and each of these entries must be assigned to a unique computer process as a seed with which to compute their local orthogonal expansions independently.

4.3 Correlation Learning

Recall that the expansion correlation matrices of a processing unit $(\mathbf{D}, \mathbf{C})_{n,m}$ are updated with bipolar pair $(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is n-dimensional and $\mathbf{y}$ is m-dimensional, via the following learning rule:

$$\mathbf{D} \leftarrow \mathbf{D} + \mathbf{y}(\Phi(\mathbf{x}))^T,$$

$$\mathbf{C} \leftarrow \mathbf{C} + (\Phi(\mathbf{x}))^T.$$

This is straightforward matrix algebra which is readily parallelizable providing the orthogonal expansion is available. Since the orthogonal expansions are permuted and locally stored according to the number of parallel processes in use, some equal liberty must be practiced in assigning the local portions of the expansion correlation matrices to the computer processes. Fortunately, this issue is somewhat negligible

providing any processing units in saved states are always loaded onto the same number of parallel processes as before, or otherwise a blank processing unit is used for each program run. Since the expansion correlation matrices $\mathbf{D}$ and $\mathbf{C}$ are zero prior to any learning, it can be assumed that each parallel process already possesses its correct local portions of these matrices that align with their local portions of the orthogonal expansion vectors. Thus, each of p parallel process should construct a local zero matrix `l_D` of dimension $m \times \frac{2^n}{p}$ and a local zero vector `l_C` of dimension $\frac{2^n}{p}$. The following is a portable parallel program for updating the local expansion correlation matrices `l_D` and `l_C`, but the algebraic operations could also be appropriately handled by calls to a linear algebra software library with some tweaking to the datatypes of `l_D` and `l_C`.

```
void parallelLearning(
        int inputSize,
        int outputSize,
        int l_expansionSize,
        int input[inputSize],
        int output[outputSize],
        int l_D[outputSize][l_expansionSize],
        int l_C[l_expansionSize],
        int processId,
        int numProcesses)
{
    int inputIndex;
    int outputIndex;
    int l_expansionIndex;
    int l_expansion[l_expansionSize];

    parallelOrthogonalExpansion(inputSize, input, l_expansion,
            processId, numProcesses);
    for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
            l_expansionIndex++) {
        l_C[l_expansionIndex] += l_expansion[l_expansionIndex];
    }

    for (outputIndex = 0; outputIndex < outputSize; outputIndex++) {
        for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
                l_expansionIndex++) {
            l_D[outputIndex][l_expansionIndex] +=
                    output[outputIndex] * l_expansion[l_expansionIndex];
        }
    }
}
```

4.4 Masking Matrix

As observed in Definition 5, the masking matrix is a diagonal matrix whose diagonal consists of a linear combination of several orthogonal expansions called entry maskings. Only the diagonal values need to be stored, and the masking matrix can be readily parallelized upon the earlier parallelization of the orthogonal expansion. The masking matrix construction requires knowledge of which orthogonal expansions must be included along its diagonal. Recall that the original masking matrix definition comprised of all combinations of entry maskings up to some specified masking level. Construction of these entry maskings amounts to looping through each set within of the power set $P(\{1, 2, \ldots, n\})$ of size less than or equal to the masking level, and using these sets as index sets with which to set particular entries in $\mathbf{e}$ to be zero. Each of the resulting vectors is then orthogonally expanded and summed with some specified weight multiple. We implement the typical entry masking weights $\alpha_{i_1,i_2,\ldots,i_j} = 2^{-wj}$ where j is the masking level index and w is a specified exponential weight. The following is a parallel C program based on this masking matrix construction. Each computer process is expected to possess a local portion of the masking matrix, denoted `l_M`.

```
void parallelMaskingMatrix(
        int inputSize,
        int l_expansionSize,
        double l_M[l_expansionSize],
        int levelSize,
        double weight,
        int processId,
        int numProcesses)
{
    int inputIndex;
    int l_expansionIndex;
    int input[inputSize];
    int l_expansion[l_expansionSize];
    int zeroIndices[levelSize + 1];
    int levelIndex = 0;
```

```
    for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
            l_expansionIndex++) {
        l_M[l_expansionIndex] = 1.0;
    }

    for (inputIndex = 0; inputIndex < inputSize; inputIndex++) {
        input[inputIndex] = 1;
    }

    levelWeights[0] = 1.0;
    levelWeights[1] = pow(0.5, ((double)weight));
    for (levelIndex = 2; levelIndex < (levelSize + 1); levelIndex++) {
        levelWeights[levelIndex] = levelWeights[1] * levelWeights[levelIndex-1];
    }

    levelIndex = 0;
    zeroIndices[0] = -1;
    while ((levelIndex > 1) || (zeroIndices[levelIndex] < (inputSize - 1))) {
        if (zeroIndices[levelIndex] < (inputSize - 1)) {
            if (levelIndex < levelSize) {
                levelIndex++;
                zeroIndices[levelIndex] = zeroIndices[levelIndex - 1] + 1;
            } else {
                zeroIndices[levelIndex]++;
            }
        } else {
            levelIndex--;
            zeroIndices[levelIndex]++;
        }
        for (inputIndex = 1; inputIndex < levelIndex + 1; inputIndex++) {
            input[zeroIndices[inputIndex]] = 0;
        }
        parallelOrthogonalExpansion(inputSize, input, l_expansion,
                processId, numProcesses);
        for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
                l_expansionIndex++) {
            l_M[l_expansionIndex] += levelWeights[levelIndex]
                    * ((double)l_expansion[l_expansionIndex]);
        }
        for (inputIndex = 1; inputIndex < levelIndex + 1; inputIndex++) {
            input[zeroIndices[inputIndex]] = 1;
        }
    }
}
```

In the program, the `while` loop steps through each combination of entry maskings up to a maximum masking level of `levelSize` provided as input. A vector of indices, `zeroIndices`, records the current combination of `input` entries which should be set to 0. When this is achieved, each process constructs a local orthogonal expansion `l_expansion` of vector `input` and applies a specified scalar multiple while adding it to the local masking matrix diagonal. The `while` loop is broken when there are no remaining entry maskings which are less than or equal to the specified

masking level. Note that this program can be slightly tweaked to instead produce a flipping matrix by setting `indices` to -1 instead of 0 on the indices contained in `zeroIndices`.

If there is need of a masking matrix with masking level $l = n$, then the construction method discussed in Section 3.3 should be utilized instead because of the preferable algorithmic complexity. The following is a parallel C program for performing this. We implement the typical entry masking weight $\alpha_i = 2^{-w}$ for all i where w is again a specified exponential weight.

```
void parallelFullMaskingMatrix(
        int inputSize,
        int l_expansionSize,
        double l_M[l_expansionSize],
        double weight,
        int processId,
        int numProcesses)
{
    int inputIndex;
    int l_expansionIndex;
    int input[inputSize];
    int l_expansion[l_expansionSize];
    double levelWeight = pow(0.5, ((double)weight));

    for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
            l_expansionIndex++) {
        l_M[l_expansionIndex] = 1.0;
    }

    for (inputIndex = 0; inputIndex < inputSize; inputIndex++) {
        input[inputIndex] = 1;
    }

    for (inputIndex = 0; inputIndex < inputSize; inputIndex++) {
        input[inputIndex] = 0;
        parallelOrthogonalExpansion(inputSize, input, l_expansion,
                processId, numProcesses);
        for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
                l_expansionIndex++) {
            l_M[l_expansionIndex] *= (1.0 +
                    levelWeight * ((double)l_expansion[l_expansionIndex]));
        }
        input[inputIndex] = 1;
    }
}
```

This program is structurally similar to the previous masking matrix program, but it is simpler since only n single entry maskings need to be produced. Thus, the third `for` loop simply replaces one entry in vector `input` with 0, produces

its orthogonal expansion, and updates the local masking matrix with the formula specified in Section 3.3. This is repeated until all entries in `input` have been replaced with 0, one at a time. Like the previous masking matrix program, this can be slightly altered to instead produce a flipping matrix by replacing each entry in `input` with -1 rather than 0 in each iteration of the third `for` loop.

4.5 Empirical Probability

As with the previous functions, the empirical probability can be readily implemented as a parallel program due to its components mostly consisting of matrix algebra, providing the parallelized implementation of the orthogonal expansion discussed earlier. Recall the formulation of the empirical probability of bipolar input $\mathbf{x}$ with masking matrix $\mathbf{M}_l$, introduced in Definition 6:

$$\rho(\mathbf{x}, \mathbf{M}_l) = \begin{cases} \frac{1}{2}\left(\frac{\mathbf{DM}_l\Phi(\mathbf{x})}{\mathbf{CM}_l\Phi(\mathbf{x})} + \mathbf{e}\right) & \text{if } \mathbf{CM}_l\Phi(\mathbf{x}) \neq 0, \\ \frac{1}{2}\mathbf{e} & \text{if } \mathbf{CM}_l\Phi(\mathbf{x}) = 0. \end{cases}$$

The following is a parallel C program implementation for producing the empirical probability of a target input.

```
void parallelEmpiricalProbability(
        int inputSize,
        int outputSize,
        int l_expansionSize,
        int input[inputSize],
        double output[outputSize],
        double l_M[l_expansionSize],
        int l_D[outputSize][l_expansionSize],
        int l_C[l_expansionSize],
        int processId,
        int numProcesses)
{
    int inputIndex;
    int outputIndex;
    int l_expansionIndex;
    int l_expansion[l_expansionSize];
    double p_c;
    double c;
```

```
    double p_output[outputSize];

    p_c = 0.0;
    for (outputIndex = 0; outputIndex < outputSize; outputIndex++) {
        p_output[outputIndex] = 0.0;
    }

    parallelOrthogonalExpansion(inputSize, input, l_expansion,
            processId, numProcesses);
    for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
            l_expansionIndex++) {
        p_c += l_M[l_expansionIndex] * ((double)(l_C[l_expansionIndex]
                * l_expansion[l_expansionIndex]));
    }
    MPI_Allreduce(&p_c, &c, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    if (c < pow(0.5, ((double)inputSize))) {
        for (outputIndex = 0; outputIndex < outputSize; outputIndex++) {
            output[outputIndex] = 0.5;
        }
    } else {
        for (outputIndex = 0; outputIndex < outputSize; outputIndex++) {
            for (l_expansionIndex = 0; l_expansionIndex < l_expansionSize;
                    l_expansionIndex++) {
                p_output[outputIndex] += l_M[l_expansionIndex]
                        * ((double)(l_D[outputIndex][l_expansionIndex]
                        * l_expansion[l_expansionIndex]));
            }
        }
        MPI_Allreduce(p_output, output, outputSize,
                MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        for (outputIndex = 0; outputIndex < outputSize; outputIndex++) {
            output[outputIndex] = 0.5 * (output[outputIndex] / c + 1.0);
        }
    }
}
```

In this implementation, the value of $\mathbf{CM}_l\Phi(\mathbf{x})$ is computed first in order to determine whether or not $\mathbf{DM}_l\Phi(\mathbf{x})$ must be computed. Each parallel process computes a partial value of $\mathbf{CM}_l\Phi(\mathbf{x})$ using its local variables `l_C`, `l_M`, and `l_expansion`, which is then stored in `p_c`. The entire value is the sum of all the `p_c` variables possessed by each parallel process, so a call to `MPI_Allreduce` is used to compute and store this sum in variable `c`. If `c` is 0, then the empirical probability vector `output` should simply contain 0.5 in each of its entries. Otherwise, $\mathbf{DM}_l\Phi(\mathbf{x})$ must be computed next. This is computed in a manner similar to determining `c`, but a vector of dimension m is produced rather than a scalar. The vector `p_output` is used to store the partial values produced of the product of the local variables `l_D`,

`l_M`, and `l_expansion`. The entire value is the sum of all the `p_output` variables possessed by each parallel process, so another call to `MPI_Allreduce` is used to compute and store this sum in variable `output`. Vector `output` is then algebraically adjusted to reflect the remaining structure of the empirical probability formula.

4.6 Performance Studies on maya with Intel MPI

A study was conducted to examine the scalability of the parallel processing unit implementation discussed thoroughly in this chapter. The purpose of this is to inspect the utility of this parallel program via the speedup and efficiency observed in employing more parallel processes to execute the processing unit functions on fixed input sizes. This study was performed on the maya cluster of the UMBC High Performance Computing Facility. Similar scalability studies on other problems have been reported [10, 9, 19, 6], and these were used as guides with which to conduct this study on the parallelized processing unit.

The UMBC High Performance Computing Facility (HPCF) is the community-based, interdisciplinary core facility for scientific computing and research on parallel algorithms at UMBC. Started in 2008 by more than 20 researchers from ten academic departments and research centers from all three colleges, it is supported by faculty contributions, federal grants, and the UMBC administration. The facility is open to UMBC researchers at no charge. Researchers can contribute funding for long-term priority access. System administration is provided by the UMBC Division of Information Technology, and users have access to consulting support provided by

dedicated full-time graduate assistants. See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

The current machine in HPCF is the distributed-memory cluster maya with over 300 nodes. The newest components of the cluster are the 72 nodes with two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory that include 19 hybrid nodes with two high-end NVIDIA K20 GPUs (graphics processing units) designed for scientific computing and 19 hybrid nodes with two cutting-edge 60-core Intel Xeon Phi 5110P accelerators. These new nodes are connected along with the 84 nodes with two quad-core 2.6 GHz Intel Nehalem X5550 CPUs and 24 GB memory by a high-speed quad-data rate (QDR) InfiniBand network for research on parallel algorithms. The remaining 168 nodes with two quad-core 2.8 GHz Intel Nehalem X5560 CPUs and 24 GB memory are designed for fastest number crunching and connected by a dual-data rate (DDR) InfiniBand network. All nodes are connected via InfiniBand to a central storage of more than 750 TB.

All results are based on the Intel implementation of MPI. Table 4.1 lists the wall clock times (a), observed speedup (b), and observed efficiency (c) measured with a parallel processing unit performing correlation learning on 10,000 training instances for fixed input size n and number of parallel processes p used. The values of n ranged from 26 to 30 in increments of 1, and the values of p ranged from 1 to 512, doubling with each increment. The compute nodes of the maya cluster used in this study each comprise two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory. Wall clock times pertaining to 1, 2, 4, 8, and 16 processes were all obtained from a single cluster node, whereas times pertaining to greater process

usage were obtained from multiple cluster nodes, exclusively using all 16 processes per node. If $T_p(n)$ denotes the wall clock time of a performance study conducted with input size n and p parallel processes, then quantity $S_p(n) = T_1(n)/T_p(n)$ measures the speedup of the program from 1 to p processes. The optimal speedup is $S_p(n) = p$, or plainly that a program of a fixed problem size using p parallel processes should ideally be p times as fast. The efficiency $E_p(n) = S_p(n)/p$ measures the closeness of the speedup to the optimum value, which itself is optimal when $E_p(n) = 1$. This optimal behavior of speedup and efficiency for a fixed problem size is known as strong scalability of parallel code.

Comparing most of the adjacent columns in the wall clock time data of Table 4.1 (a) indicates that utilizing twice as many parallel processors roughly reduces the run time by a factor of two, except notably when comparing columns $p = 4$, $p = 8$, and $p = 16$. These particular wall clock times cause the overall trend of the observed speedup S_p to deviate significantly from the optimal value $S_p(n) = p$. This is recorded in Table 4.1 (b), which lists the respective observed speedup from 1 to p processes for each problem size. Figure 4.1 visualizes the observed speedup and efficiency to more easily convey comparisons between these quantities and their optimal values. The suboptimal speedup that takes occurs in $p = 4$, $p = 8$, and $p = 16$ can be attributed to a few factors. Foremost, the orthogonal expansion program is not fully parallelized since a sufficient amount of the expansion must be constructed in serial so that each parallel process may have an assigned seed value with which to compute their remaining local portions of the expansion. Additionally, the orthogonal expansion program may not make effective use of the

CPU cache since its recursive nature requires several revisits to the earlier expansion entries. These shortcomings become exaggerated when training over 10,000 data instances, with each rendition of correlation learning requiring the construction of a new orthogonal expansion. Note that the run times for 32, 64, 128, and 256 processes return to the ideal factor of two speedup when comparing the adjacent columns. This can be attributed to the greater availability of CPU cache memory by means of utilizing more computing nodes. The distribution of labor onto multiple compute nodes produces local problems which more easily fit into the cache, leading to fewer cache misses and improved run time. Despite the more ideal performance for 32, 64, and 256 processes, the overall observed speedups and efficiencies (c) remain suboptimal due to the portion of the orthogonal expansion program which could not be parallelized. Although the observed speedups and efficiencies are far from ideal overall, the parallelized processing unit program retains value in dramatically reducing the run time from several hours to a few minutes for large input sizes when utilizing a sufficient amount of parallel processes.

The performance study is repeated in Table 4.2 and accompanying Figure 4.2 in which the scalability of the empirical probability function is examined over 10,000 target vectors. Comparing these results with those of correlation learning in Table 4.1, it is evident that the parallelized empirical probability program suffers from similar issues with suboptimal speedup and efficiency due to the orthogonal expansion program. Again, a rough speedup factor of two can be observed when comparing adjacent wall clock time columns for 32, 64, 128, 256, and 512 processes, but this is not the case for 4, 8, and 16 processes which hinder the overall observed speedup and

efficiency. Construction of the empirical probability generally takes more time to perform than correlation learning due to the greater amount computation involved and, more substantially, the fact that this function requires communication among the processes whereas correlation learning does not. This is unavoidable since the local variables must be combined to produce a coherent empirical probability vector with which to generate label predictions. Despite this and the suboptimal observations of speedup and efficiency, it remains evident that the parallelized processing unit retains value in dramatically reducing the run time of sufficiently large problems from several hours to a few minutes with the utilization of more parallel processes.

Table 4.1: Intel MPI correlation learning performance on maya by number of processes used with 16 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, $p = 4$ which uses 4 processes per node, and $p = 8$ which uses 8 processes per node.

(a) Wall clock time in HH:MM:SS

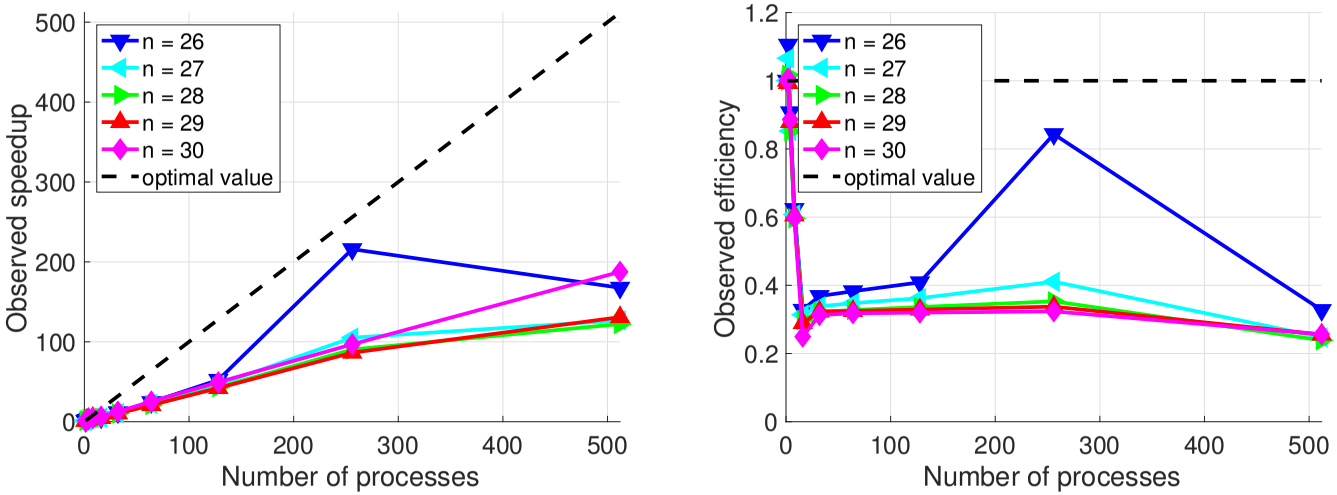
n	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
26	00:26:51	00:12:08	00:07:24	00:05:22	00:05:07	00:02:16	00:01:05	00:00:30	00:00:07	00:00:09
27	00:50:32	00:23:42	00:14:49	00:10:23	00:10:04	00:04:40	00:02:16	00:01:05	00:00:28	00:00:23
28	01:37:43	00:48:01	00:28:41	00:20:28	00:20:30	00:09:28	00:04:39	00:02:16	00:01:04	00:00:47
29	03:16:15	01:38:46	00:55:50	00:40:32	00:42:25	00:18:59	00:09:26	00:04:39	00:02:16	00:01:29
30	06:26:54	03:12:19	01:49:09	01:20:52	01:36:58	00:38:39	00:19:00	00:09:26	00:04:40	00:02:57

(b) Observed speedup S_p

n	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
26	1.0000	2.2114	3.6296	4.9896	5.2492	11.7937	24.4825	52.3244	215.9116	167.6325
27	1.0000	2.1320	3.4108	4.8625	5.0196	10.8261	22.2529	46.3666	105.1547	126.6857
28	1.0000	2.0345	3.4061	4.7710	4.7664	10.3123	20.9641	43.0604	90.3753	122.2653
29	1.0000	1.9867	3.5150	4.8405	4.6258	10.3342	20.7897	42.1118	86.3733	130.8900
30	1.0000	2.0117	3.5446	4.7843	3.9895	10.0097	20.3528	40.9492	82.8559	131.1388

(c) Observed efficiency E_p

n	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
26	1.0000	1.1057	0.9074	0.6237	0.3281	0.3686	0.3825	0.4088	0.8434	0.3274
27	1.0000	1.0660	0.8527	0.6078	0.3137	0.3383	0.3477	0.3622	0.4108	0.2474
28	1.0000	1.0172	0.8515	0.5964	0.2979	0.3223	0.3276	0.3364	0.3530	0.2388
29	1.0000	0.9934	0.8787	0.6051	0.2891	0.3229	0.3248	0.3290	0.3374	0.2556
30	1.0000	1.0059	0.8862	0.5980	0.2493	0.3128	0.3180	0.3199	0.3237	0.2561



(a) Observed speedup S_p

(b) Observed efficiency E_p

Figure 4.1: Intel MPI correlation learning performance on maya by number of processes used with 16 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, $p = 4$ which uses 4 processes per node, and $p = 8$ which uses 8 processes per node.

Table 4.2: Intel MPI empirical probability performance on maya by number of processes used with 16 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, $p = 4$ which uses 4 processes per node, and $p = 8$ which uses 8 processes per node.

(a) Wall clock time in HH:MM:SS

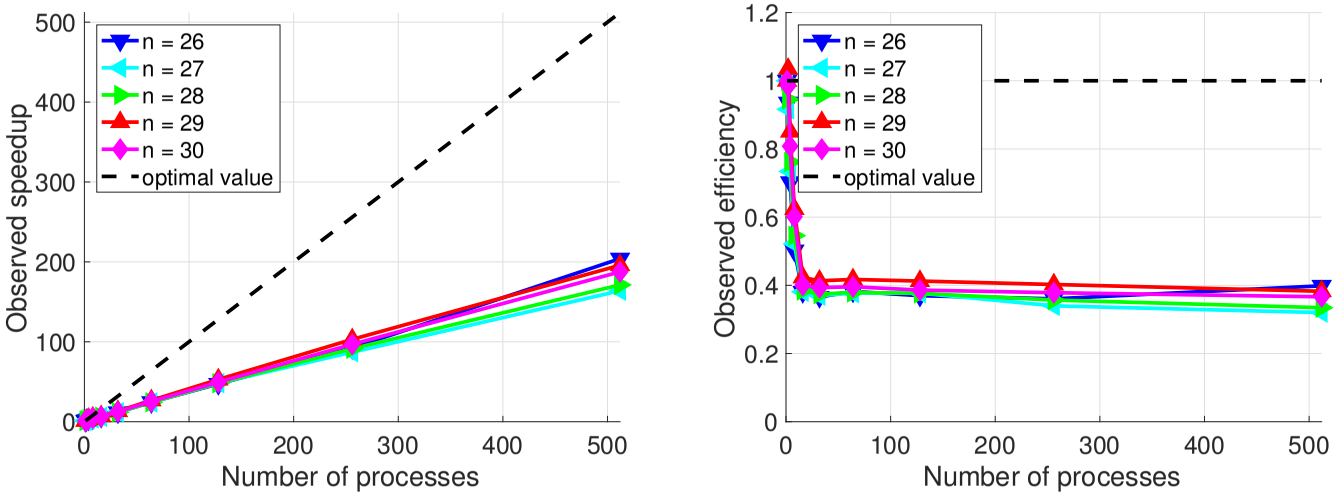
n	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
26	00:32:29	00:17:21	00:11:33	00:08:05	00:05:20	00:02:46	00:01:19	00:00:41	00:00:21	00:00:09
27	01:05:34	00:35:47	00:22:19	00:15:42	00:10:45	00:05:32	00:02:42	00:01:21	00:00:45	00:00:24
28	02:12:42	01:10:13	00:43:33	00:30:23	00:21:34	00:11:04	00:05:27	00:02:45	00:01:27	00:00:46
29	04:52:29	02:21:20	01:25:57	00:58:36	00:43:12	00:22:07	00:10:57	00:05:32	00:02:50	00:01:29
30	09:13:44	04:41:02	02:51:18	01:55:10	01:26:28	00:44:00	00:21:51	00:11:12	00:05:43	00:02:57

(b) Observed speedup S_p

n	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
26	1.0000	1.8716	2.8093	4.0163	6.0875	11.7119	24.4503	47.3110	92.2727	204.0412
27	1.0000	1.8325	2.9365	4.1735	6.0964	11.8291	24.1479	48.2744	86.9932	163.8708
28	1.0000	1.8899	3.0468	4.3665	6.1525	11.9754	24.2759	48.1768	91.3495	171.2178
29	1.0000	2.0694	3.4028	4.9898	6.7704	13.2233	26.7049	52.8025	103.0321	195.8654
30	1.0000	1.9703	3.2324	4.8076	6.4034	12.5845	25.3281	49.4073	96.8583	187.5704

(c) Observed efficiency E_p

n	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$	$p = 128$	$p = 256$	$p = 512$
26	1.0000	0.9358	0.7023	0.5020	0.3805	0.3660	0.3820	0.3696	0.3604	0.3985
27	1.0000	0.9163	0.7341	0.5217	0.3810	0.3697	0.3773	0.3771	0.3398	0.3201
28	1.0000	0.9449	0.7617	0.5458	0.3845	0.3742	0.3793	0.3764	0.3568	0.3344
29	1.0000	1.0347	0.8507	0.6237	0.4231	0.4132	0.4173	0.4125	0.4025	0.3825
30	1.0000	0.9852	0.8081	0.6010	0.4002	0.3933	0.3958	0.3860	0.3784	0.3663



(a) Observed speedup S_p

(b) Observed efficiency E_p

Figure 4.2: Intel MPI empirical probability performance on maya by number of processes used with 16 processes per node, except for $p = 1$ which uses 1 process per node, $p = 2$ which uses 2 processes per node, $p = 4$ which uses 4 processes per node, and $p = 8$ which uses 8 processes per node.

4.7 Digit Recognition Example

In this section, the performance of the processing unit program in pattern recognition is demonstrated via a small example involving visual depictions of the ten numerical digits. The digits can be represented with 5×3 bipolar matrices in which 1 is used to "write" the digit and -1 denotes blank space. Figure 4.3 displays the bipolar representations of each numeral as a color map, imitating the digits as they appear on common digital clocks. For demonstration, a processing unit is trained on these ten matrices provided as vectors in column-major ordering. The associated output are 10-dimensional bipolar vectors in which each entry corresponds with the identity of the digit from 0 through 9. So the matrix depiction of zero would be learned with an output vector consisting of all negative values except for a positive value in the first entry. This manner of output encoding will produce empirical probability vectors in which each entry denotes the probability that the target input is identified as the corresponding numeral.

Table 4.3 lists the empirical probability vectors produced on each of the ten 5×3 bipolar visualizations of the numerical digits. The processing unit had prior training to recognize each digit, and the empirical probabilities are produced using masking matrix generalization with maximum masking level $l = 15$ and masking level weights 2^{-wj} where $j = 0, 1, 2, \ldots, l$. For comparison purposes, hyperparameter w is set to zero so that all masking level weights are effectively 1. Each row of Table 4.3 corresponds with the target input that is provided for empirical probability construction, and each column corresponds with an entry in the empirical probability

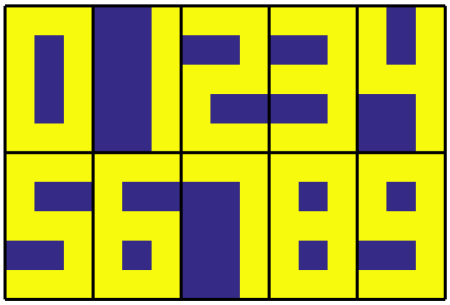


Figure 4.3: Color map of the numerical digits visually represented in 5×3 bipolar matrices.

vector respective to each of the ten digit classes. For example, the input depicting 0 has probability 0.5969 of being correctly classified as numeral 0 and probability 0.0003 of being incorrectly classified as numeral 1. Based on the table organization, probabilities of correct classification are along the table diagonal and all other entries are probabilities of incorrect classifications. Overall, the diagonal entries are greater in magnitude when compared with other entries in their respective rows, but these other entries clearly indicate some uncertainty which occurs naturally with entry masking generalization. Recall that entry masking attempts to ignore distortions that exist between the target input vector and the set of learned vectors in order to generalize. The masking matrix inherently favors lower level entry maskings without the need for masking level weights, effectively giving more weight to learned vectors which are nearer in Hamming distance to the target vector. This role of Hamming distance can be clearly observed in the empirical probabilities along the columns of Table 4.3. For example, the second highest entry for row 0 is 0.1990 corresponding

with the digit class 8. This being the second highest entry is sensible because the 5×3 bipolar inputs depicting 0 and 8 are one unit of Hamming distance away from each other, whereas the distances between the visualizations of 0 and all the other digits are greater than 1. Regardless, it would be preferable to push the probabilities of correct classification closer to 1 and all others closer to 0.

Table 4.4 repeats the correlation learning and empirical probability construction on the 5×3 bipolar digit visualizations, this time with hyperparameter $w = 10$. This increase in w effectively increases the weight of lower level entry maskings relative to higher level entry maskings. As anticipated, this produces the desired result of increasing the probabilities of correct classifications seen along the table diagonal and decreasing all other probabilities associated with incorrect classifications. Small errors remain, such as the 0.0005 probability of incorrectly classifying input 0 as digit 8. These erroneous probabilities can be further decreased by further increasing w.

Table 4.3: Empirical probabilities produced by a processing unit trained on example digit recognition data with entry masking hyperparameters $l = 15$ and $w = 0$.

actual	predicted 0	1	2	3	4	5	6	7	8	9
0	0.5969	0.0003	0.0221	0.0221	0.0025	0.0221	0.0663	0.0025	0.1990	0.0663
1	0.0004	0.8879	0.0001	0.0012	0.0110	0.0001	0.0000	0.0987	0.0001	0.0004
2	0.0275	0.0001	0.7415	0.0824	0.0010	0.0092	0.0275	0.0010	0.0824	0.0275
3	0.0210	0.0008	0.0629	0.5660	0.0070	0.0629	0.0210	0.0070	0.0629	0.1887
4	0.0037	0.0111	0.0012	0.0111	0.9022	0.0111	0.0037	0.0111	0.0111	0.0334
5	0.0190	0.0001	0.0063	0.0569	0.0063	0.5123	0.1708	0.0007	0.0569	0.1708
6	0.0565	0.0000	0.0188	0.0188	0.0021	0.1694	0.5082	0.0002	0.1694	0.0565
7	0.0036	0.0967	0.0012	0.0107	0.0107	0.0012	0.0004	0.8706	0.0012	0.0036
8	0.1420	0.0001	0.0473	0.0473	0.0053	0.0473	0.1420	0.0006	0.4260	0.1420
9	0.0483	0.0002	0.0161	0.1449	0.0161	0.1449	0.0483	0.0018	0.1449	0.4346

Table 4.4: Empirical probabilities produced by a processing unit trained on example digit recognition data with entry masking hyperparameters $l = 15$ and $w = 10$.

actual	predicted 0	1	2	3	4	5	6	7	8	9
0	0.9995	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0005	0.0000
1	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
2	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
3	0.0000	0.0000	0.0000	0.9995	0.0000	0.0000	0.0000	0.0000	0.0000	0.0005
4	0.0000	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000
5	0.0000	0.0000	0.0000	0.0000	0.0000	0.9990	0.0005	0.0000	0.0000	0.0005
6	0.0000	0.0000	0.0000	0.0000	0.0000	0.0005	0.9990	0.0000	0.0005	0.0000
7	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000
8	0.0005	0.0000	0.0000	0.0000	0.0000	0.0000	0.0005	0.0000	0.9985	0.0005
9	0.0000	0.0000	0.0000	0.0005	0.0000	0.0005	0.0000	0.0000	0.0005	0.9985

4.8 Summary

A parallel implementation of the THPAM processing unit architecture was proposed and thoroughly discussed in this chapter. The processing unit was implemented as a C program with MPI for communication between parallel processes. The processing unit correlation learning and empirical probability mechanisms mostly involve matrix algebra computations which are readily parallelizable. However, all of the processing unit mechanisms rely on the construction of the orthogonal expansions of input vectors. One method of parallelizing the orthogonal expansion was proposed, but this was not fully parallelizable and required some creative liberty in distributing the orthogonal expansion entries among the parallel processes. Unfortunately, the unparallelized portion of the orthogonal expansion requires computation proportional to the number of parallel processes in use, since a sufficient amount of the expansion must be constructed to assign an initial seed value to each parallel process with which to construct the remaining local expansion entries.

A scalability study of the learning and decovariance retrieval mechanisms was conducted on the UMBC HPCF maya cluster. The observed speedup and efficiency from 1 to p processes were determined to be suboptimal due to the lack of a fully parallelized orthogonal expansion procedure, but parallel processing still dramatically reduces the run time of these processing unit mechanisms for sufficiently large input sizes. Although the exponential complexity of the orthogonal expansion imposes a severe limitation on the input size, parallel computing allows us to push the input

size beyond what would be acceptable with a serial computer. Lastly, the capability of the processing unit as a pattern recognizer was demonstrated on a small example involving visual depictions of the numerical digits.

Chapter 5

Experiments

5.1 Overview

This chapter includes reports on several experiments used to examine the performance of the THPAM processing unit architecture in classification. The data sets utilized in these experiments were obtained from the University of California Irvine (UCI) Machine Learning Repository [12], which hosts numerous databases and research tools for the empirical analysis of learning algorithms. For each data set featured in this work, multiple data encoding methods are used in order to present the data in a manner suitable for processing by the processing unit. Some encoding methods are more appropriate than others, and these reports attempt to offer explanations and recommendations for best practices. At the time of this writing, computational experiments on the THPAM processing unit have not yet been published, so this work also serves as an initial benchmark as this learning algorithm becomes more theoretically mature. These experiments provide key insights that may prove useful in future work.

The considered data sets are not provided with distinct training and test subsets, and there is no official standard for evaluating algorithm performance on these data sets. Leave-one-out cross-validation is selected to measure the performance of the processing unit on each data set and with each examined data set encoding. In

this measure, the processing unit is trained on the entire data set except for one data instance on which the predictive accuracy is tested. This procedure is repeated until all of the data instances have been used for testing and a final accuracy rating can thereby be computed. This measure is selected because it provides a complete result with which the processing unit performance can be definitively compared on the different encoding techniques examined herein. The supervised processing unit constructed for these studies is deterministic in nature, providing some specified rules on breaking ties in classification. It is also inexpensive to perform leave-one-out cross-validation using this supervised processing unit since data instances previously learned by the processing unit can be easily removed for validation and subsequently relearned. This is generally untrue for most learning algorithms, with which leave-one-out cross-validation is actually very expensive to perform in relation to other validation measures.

In general, k-fold cross-validation or repeated k-fold cross-validation are used in related publications, where k is usually set to 5 or 10. In addition to examining the processing unit performance with varying encoding methods and hyperparameter values, select repeated stratified 10-fold cross-validation results are also reported for each of the data sets considered herein. With some historical 10-fold cross-validation results reported on other known learning algorithms, we can observe how well the processing unit measures up to the competition at this early stage in its realization as a learning machine. For each data set considered herein, the results of other learning algorithms reported in some related publications are summarized for the purpose of comparison.

Section 5.2 reports the processing unit performance on a categorical data set with three different input encoding methods to examine how category rank can be interpreted by the processing unit via input vectors. In Section 5.3, the performance of the processing unit is tested on a binary data set with missing values. Section 5.4 describes an encoding method for real-valued data, with which the processing unit performance can be improved via a custom entry masking weighting strategy to account for the relative significance of the input entries.

5.2 UCI Car Evaluation Data Set

The UCI Car Evaluation data set [2] consists of categorical measures of car acceptability based on pricing and technical characteristics. Each car is evaluated according to six categorical features listed in Table 5.1 with their respective potential values. The data set is exhaustive, consisting of 1728 unique data instances which cover all possible input combinations within the feature space. Car acceptability is labeled with four possible classes listed in Table 5.2 alongside their respective class distributions among the data instances. The classes are unevenly represented with the majority class constituting 70.0231% of the entire data set. This should be considered the lowest bound for the predictive accuracy of a learning algorithm trained on this data, since a rudimentary algorithm could blindly guess the majority class to achieve this performance.

In order to apply the THPAM processing unit to this categorical database, the feature inputs and class outputs must be properly encoded into bipolar vectors

suitable for processing. Herein, we investigate three candidate encoding techniques while also examining the impact of hyperparameter adjustments in the competing generalization methods of entry masking and entry flipping. The performance metric utilized in these experiments is leave-one-out cross-validation. Class labels are encoded as one-hot bipolar vectors, where each class is assigned a distinct output entry which is positive for a given data instance if and only if that instance belongs to the corresponding class. Class predictions are generated by producing the empirical probability then selecting the class associated with the entry that has largest probability relative to the other entries. Any ties are broken by selecting the class which has greater representation in the instance data. For example, if the empirical probability of a target input vector indicates a classification of the `acceptable` and `good` classes in equal measure, then `acceptable` is selected as the prediction since this class has greater distribution among the data instances.

Table 5.1: Car Evaluation data features and their respective categorical values.

Feature	Values
buying price	very high, high, medium, low
maintenance cost	very high, high, medium, low
number of doors	2, 3, 4, 5 or more
occupancy	2, 4, more
luggage boot size	small, medium, big
safety	low, medium, high

Table 5.2: Car Evaluation data classes and their respective distributions among the instance data.

Class	# of instances	% of instances
unacceptable	1210	70.0231%
acceptable	384	22.2222%
good	69	3.9931%
very good	65	3.7616%
total	1728	

5.2.1 Bipolar Encoding

We first examine a commonly used categorical data encoding technique which simply attempts to uniquely represent all the possible feature values with the fewest number of binary or bipolar entries. For example, a feature which has 4 possible values requires at least $\log_2(4) = 2$ entries to be fully encoded. This is a straightforward method which imposes the least demand on computer resources, but it can inadequately account for any underlying relationships between the feature values. This fact is better observed after analyzing the performance involving each of the three encoding techniques considered in these experiments on the Car Evaluation data set. For now we focus solely on this encoding technique. Table 5.3 lists the car features and their corresponding categorical values, each value paired with a bipolar vector encoding used in these experiments. Each of the six features requires at least two bipolar entries to completely represent their potential values, producing a total input size of twelve entries.

Table 5.4 lists the observed leave-one-out cross-validation accuracy ratings with varying hyperparameter adjustments pertaining to the entry masking (a) and

Table 5.3: Car Evaluation data features and their respective categorical values, each value paired with a bipolar encoding using the fewest possible bits.

Feature	Values
buying price	{very high : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {high : $\begin{pmatrix} 1 & -1 \end{pmatrix}$}, {medium : $\begin{pmatrix} -1 & 1 \end{pmatrix}$}, {low : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}
maintenance cost	{very high : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {high : $\begin{pmatrix} 1 & -1 \end{pmatrix}$}, {medium : $\begin{pmatrix} -1 & 1 \end{pmatrix}$}, {low : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}
number of doors	{5 more : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {4 : $\begin{pmatrix} 1 & -1 \end{pmatrix}$}, {3 : $\begin{pmatrix} -1 & 1 \end{pmatrix}$}, {2 : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}
occupancy	{more : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {4 : $\begin{pmatrix} 1 & -1 \end{pmatrix}$}, {2 : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}
luggage boot size	{big : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {medium : $\begin{pmatrix} 1 & -1 \end{pmatrix}$}, {small : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}
safety	{high : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {medium : $\begin{pmatrix} -1 & 1 \end{pmatrix}$}, {low : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}

entry flipping (b) generalization techniques. The table columns correspond with the masking level l, the maximum number of input entries allowed to be masked or flipped in order to relate an input vector with a learned vector when constructing the empirical probability. The table rows correspond with the exponential weight w of the entry masking weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l. Larger w produces more relative weight on lower level entry maskings than higher level maskings.

The entire column for $l = 0$ of Table 5.4 reports accuracy ratings of 70.0231%, equal to the distribution of the majority class. This is because the masking level of $l = 0$ produces masking and flipping matrices equivalent to the identity matrix, meaning no generalization is being performed. In this case, the THPAM processing

unit is simply guessing the majority class. The adjacent column for $l = 1$ allows generalization up to a hamming distance of one, which dramatically improves performance up to 95.8912%. This accuracy appears in every row because there are no learned vectors of hamming distance zero with which to relate to a given test vector. As stated in the data set description, the Car Evaluation set is exhaustive and each training instance is distinct, so a test vector held out of training would have no training instance of hamming distance zero with which to compare to. The value of hyperparameter w is thereby irrelevant at this masking level for this particular data set. The columns with $l \geq 2$ demonstrate the importance of selecting sufficiently large exponential weight w in order to counterbalance the increase in masking level, especially evident in rows with $w \leq 1$. Comparing adjacent columns of these rows indicates a steady decline towards an accuracy rating equivalent to the majority class distribution. This is expected since the increase in masking level allows for each test vector to be compared with an increasingly larger subset of the training data, eventually including all of the training data in predictions when the masking level is equivalent to the input size. An insufficiently large w fails to counterbalance the unequal representation of the majority class, giving this class more relative weight by sheer numbers. Comparing adjacent rows indicates that larger w effectively corrects this uneven distribution, giving more relative weight to learned vectors which are nearer in Hamming distance to the test vector. All masking levels of $l \geq 2$ are capable of achieving the maximum leave-one-out cross-validation accuracy of 96.1227% providing that w is large enough, as observed in the rows with $w \geq 4$ for the masking matrix and the row with $w = 5$ for the flipping

matrix. This behavior is visualized for both the masking matrix and flipping matrix generalization mechanisms in Figure 5.1 (a) and (b), respectively. These figures plot the cross-validation accuracy by masking level l for $w = 0$, $w = 1$, and $w = 6$ to more clearly demonstrate the trend toward maximum test accuracy for any l as w is made sufficiently large.

Comparing the cross-validation accuracies of entry masking generalization (a) and entry flipping generalization (b) in Table 5.4 indicates some notable differences in performance between these techniques. Masking generalization always achieves equal or greater test accuracy than entry flipping generalization for the same combination of hyperparameter values. This is due to the inherent behavior of the masking matrix to give more relative weight to learned vectors which are nearer in hamming distance to the target test vector. This is separate from the masking level weights 2^{-wj}, as demonstrated in Example 4. The inherent weighting of the masking matrix works in conjunction with the level weights, whereas the flipping matrix has no inherent weighting and must rely solely on masking level weights to achieve the desired weighting scheme. The accuracies of these generalization methods are significantly different when $w \leq 3$, with the masking matrix outperforming the flipping matrix. However, both techniques trend toward the same accuracy maximum of 96.1227% as w is made sufficiently large. Figure 5.2 visualizes this trend for both generalization mechanisms with masking level l equal to the input dimension $n = 12$. This plots the test accuracy by the exponential weight w, demonstrating that although the masking matrix outperforms the flipping matrix for small w, both mechanisms tend toward the same maximum test accuracy.

A glaring fault of this bipolar encoding of the input data is that the processing unit generalizes based on the hamming distance metric, but these encodings create erroneous hamming distance relations between the different feature categories. For example, the `buying price` feature has categories `very high`, `high`, and `medium` encoded with $(1\ 1)$, $(1\ -1)$, and $(-1\ 1)$, respectively, implying that `very high` is equidistant to `high` and `medium`. In processing unit generalization this means that, with all else equal, a test vector with `buying price` value of `very high` will be compared to learned vectors with `high` and `medium` values with equal weight, even though the vector with value `high` should sensibly have more comparative weight as it is closer in quantity to `very high`. This could be changed by rearranging the encodings, but the same problem will inevitably appear in another set of categories. Minimal feature encoding fails to capture the natural order implied by some feature categories, a point that will be more clearly conveyed in the experiments ahead involving alternative encoding schemes.

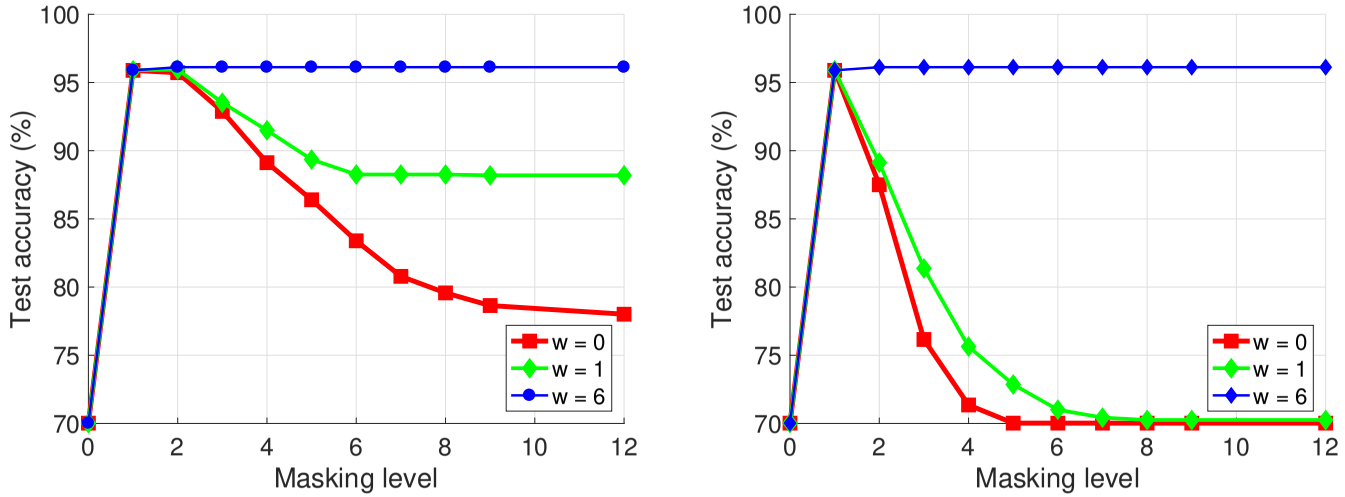
Table 5.4: Leave-one-out cross-validation accuracies on bipolar encoded Car Evaluation data by masking level and masking level weight. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

(a) Accuracy ratings with entry masking generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=12$
0	70.0231	95.8912	95.7176	92.8819	89.1204	86.4005	83.3912	80.7870	79.5718	78.6458		78.0093
1	70.0231	95.8912	95.9491	93.5185	91.4931	89.3519	88.2523	88.2523	88.2523	88.1944		88.1944
2	70.0231	95.8912	96.1227	94.2708	93.7500	93.2870	93.1713	93.1713	93.1713	93.1713		93.1713
3	70.0231	95.8912	96.1227	96.0069	95.7755	95.7755	95.7755	95.7755	95.7755	95.7755	...	95.7755
4	70.0231	95.8912	96.1227	96.1227	96.1227	96.1227	96.0069	96.0069	96.0069	96.0069		96.0069
5	70.0231	95.8912	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227		96.1227
6	70.0231	95.8912	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227		96.1227

(b) Accuracy ratings with entry flipping generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=12$
0	70.0231	95.8912	87.5000	76.1574	71.3542	70.0231	70.0231	70.0231	70.0231	70.0231		70.0231
1	70.0231	95.8912	89.1204	81.3657	75.6366	72.8588	71.0069	70.4282	70.2546	70.2546		70.2546
2	70.0231	95.8912	92.0139	87.9051	85.8796	85.0116	84.6644	84.4907	84.4329	84.4329		84.4329
3	70.0231	95.8912	94.3866	92.9398	92.4769	92.3032	92.3032	92.3032	92.3032	92.3032	...	92.3032
4	70.0231	95.8912	96.1227	95.6019	95.4282	95.4282	95.4282	95.4282	95.4282	95.4282		95.4282
5	70.0231	95.8912	96.1227	96.1227	96.0069	96.0069	96.0069	96.0069	96.0069	96.0069		96.0069
6	70.0231	95.8912	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227	96.1227		96.1227



(a) Entry masking

(b) Entry flipping

Figure 5.1: Learning curves of a processing unit applied to bipolar encoded Car Evaluation data by masking level. Accuracy ratings are shown for quantities $w = 0$, $w = 1$, and $w = 6$ with entry masking generalization (a) and entry flipping generalization (b). Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

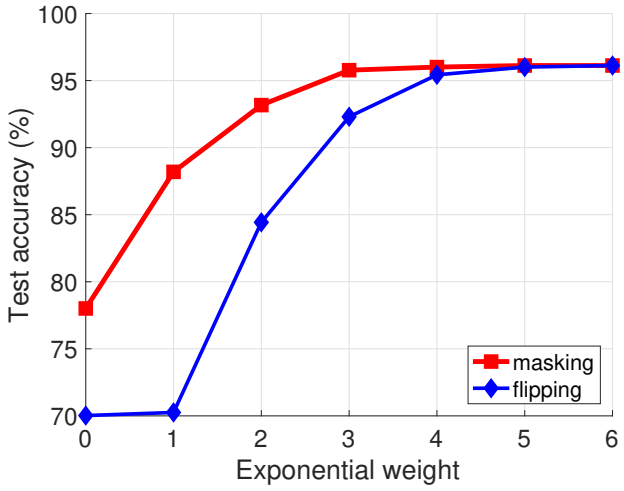


Figure 5.2: Learning curves of a processing unit applied to bipolar encoded Car Evaluation data by exponential weight w with masking level $l = 12$. Accuracy ratings are shown for entry masking generalization and entry flipping generalization. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

5.2.2 One-hot Encoding

Another commonly used categorical data encoding technique is one-hot encoding, in which each feature is encoded into a binary or bipolar vector of size equivalent to the number of categorical values of that feature. Each entry in the encoding corresponds to a particular feature value, and the entry is positive if and only if the data instance has that particular feature value. For example, the `buying price` feature values could be one-hot encoded with $(\,1\ -1\ -1\ -1\,)$ for `very high`, $(\,-1\ 1\ -1\ -1\,)$ for `high`, $(\,-1\ -1\ 1\ -1\,)$ for `medium`, and $(\,-1\ -1\ -1\ 1\,)$ for `low`. Consequently, an input size of 21 is required to fully encode the Car Evaluation features in this manner. This

encoding method requires as many input entries as there are categories among all the features in the data set, so it is the opposite of bipolar encoding in the sense that one-hot encoding demands significantly more computer memory. However, an advantage of one-hot encoding is that it entirely ignores any ordering that might exist among the different categories of a feature. The Hamming distance between any two feature categories is always 2, as observed in the example above for the `buying price` feature. As will be demonstrated experimentally, this is a slight improvement on the erroneous categorical ordering imposed by bipolar encoding, but the implied categorical ordering remains disregarded.

The leave-one-out cross-validation experiment is repeated and summarized in Table 5.5 with one-hot encoded input vectors with the entry masking (a) and entry flipping (b) generalization mechanisms. Again, the table columns and rows are parameterized by masking level l and exponential weight w, respectively. Only accuracy ratings associated with even masking level values are reported due to the fact that each one-hot encoded feature category is a Hamming distance of two away from the other feature categories. Since there are six features, the masking level should be twelve at most. As observed in the previous experiment, a masking level of $l = 0$ results in no performed generalization, forcing the processing unit to guess the majority class with 70.0231% accuracy. The column with masking level $l = 2$ records significant improvement in the processing unit predictive accuracy, with a rating of 93.6921%. This does not compete with the maximum recorded cross-validation accuracy of 96.1227% obtained in the previous experiment involving bipolar encoding. The columns with $l \geq 8$ and $l \geq 2$ for entry masking and entry

flipping, respectively, again demonstrate the importance of selecting sufficiently large exponential weight w in order to counterbalance the increase in masking level. With $w \leq 1$, a gradual decline in accuracy can be observed as l increases, due to this effectively increasing the subset of learned vectors with which the test vectors may be compared to. Thus, the accuracies along these rows tend towards the majority class distribution with entry flipping generalization, but comparatively the respective accuracies with entry masking generalization do not decline as much. This can be attributed to the inherent weighting mechanism that exists in the masking matrix and the evidential fact that one-hot input encoding appears to be better suited for the processing unit. Regardless, all masking levels of $l \geq 2$ are capable of achieving the maximum leave-one-out cross-validation accuracy of 93.6921% providing that w is large enough, as observed in the rows with $w \geq 2$ for entry masking and the rows with $w \geq 3$ for entry flipping. This trend towards the maximum observed accuracy is plotted in Figure 5.3 with entry masking (a) and entry flipping (b) generalization for $w = 0$, $w = 1$, and $w = 5$. Figure 5.4 plots a comparison of performance between the entry masking and entry flipping mechanisms, indicating again that entry masking may be preferable when w is small, but both generalization techniques achieve similar performance when w is made sufficiently large.

The implied ordering of the feature categories remains incorrectly accounted for with the one-hot feature encoded inputs. One-hot encoding assumes that all categories are equidistant from each other, so a `buying price` feature value of `very high` can be compared to values `high`, `medium`, and `low` with equal weight under generalization. The next section examines a more sensible encoding scheme

that better captures the implied ordering of these feature values.

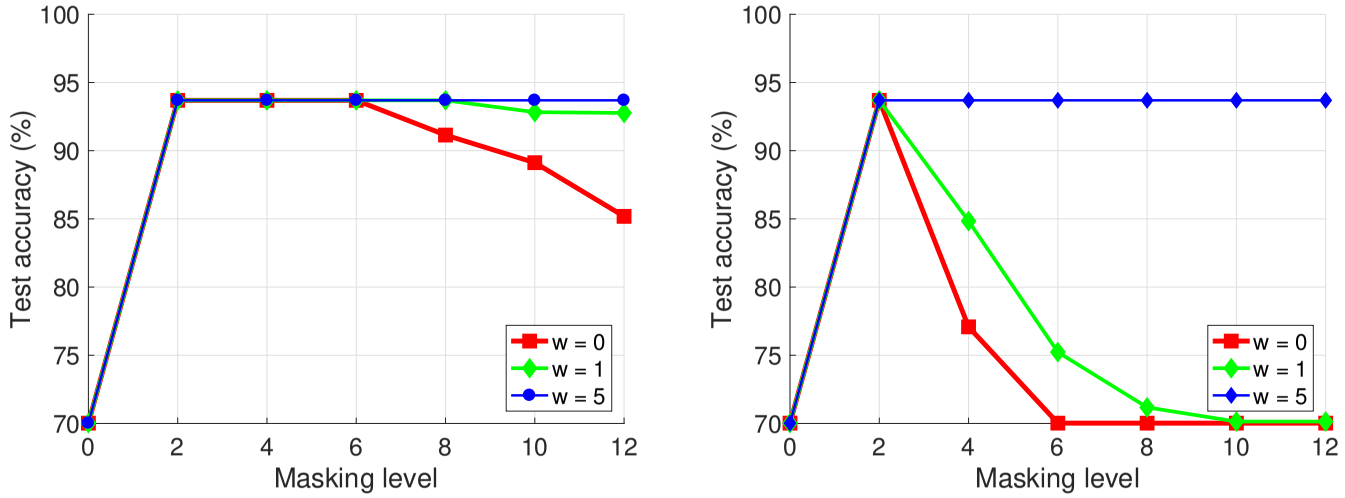
Table 5.5: Leave-one-out cross-validation accuracies on one-hot encoded Car Evaluation data by masking level and masking level weight. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

(a) Accuracy ratings with entry masking generalization

w	$l = 0$	$l = 2$	$l = 4$	$l = 6$	$l = 8$	$l = 10$	$l = 12$
0	70.0231	93.6921	93.6921	93.6921	91.1458	89.1204	85.1852
1	70.0231	93.6921	93.6921	93.6921	93.6921	92.8241	92.7662
2	70.0231	93.6921	93.6921	93.6921	93.6921	93.6921	93.6921
3	70.0231	93.6921	93.6921	93.6921	93.6921	93.6921	93.6921
4	70.0231	93.6921	93.6921	93.6921	93.6921	93.6921	93.6921
5	70.0231	93.6921	93.6921	93.6921	93.6921	93.6921	93.6921

(b) Accuracy ratings with entry flipping generalization

w	$l = 0$	$l = 2$	$l = 4$	$l = 6$	$l = 8$	$l = 10$	$l = 12$
0	70.0231	93.6921	77.0833	70.0231	70.0231	70.0231	70.0231
1	70.0231	93.6921	84.8380	75.2315	71.1806	70.1389	70.1389
2	70.0231	93.6921	91.3773	90.8565	90.7407	90.7407	90.7407
3	70.0231	93.6921	93.6921	93.6921	93.6921	93.6921	93.6921
4	70.0231	93.6921	93.6921	93.6921	93.6921	93.6921	93.6921
5	70.0231	93.6921	93.6921	93.6921	93.6921	93.6921	93.6921



(a) Entry masking

(b) Entry flipping

Figure 5.3: Learning curves of a processing unit applied to one-hot encoded Car Evaluation data by masking level. Accuracy ratings are shown for quantities $w = 0$, $w = 1$, and $w = 5$ with entry masking generalization (a) and entry flipping generalization (b). Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

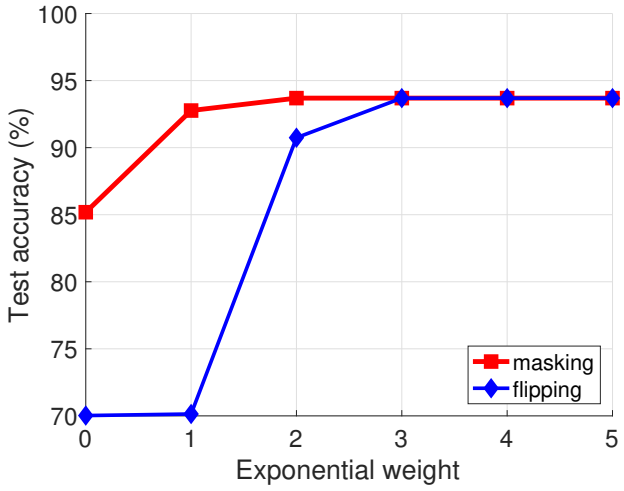


Figure 5.4: Learning curves of a processing unit applied to one-hot encoded Car Evaluation data by exponential weight w with masking level $l = 21$. Accuracy ratings are shown for entry masking generalization and entry flipping generalization. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

5.2.3 Temperature Encoding

This final encoding technique attempts to represent the feature categories in a manner which preserves their implicit ordering with respect to the Hamming distance metric [16]. To exemplify this, we return to the `buying price` feature of the Car Evaluation data set. In performing generalization on a test vector with a `buying price` of value `very high`, it is sensible to give more comparative weight to learned vectors with value `high` than those with value `medium`, and learned vectors with value `low` should receive even smaller weight. The implied order is that `low` < `medium` < `high` < `very high`, but this order was not preserved among the pairwise

hamming distances in the bipolar and one-hot encoding methods.

Table 5.6 lists the Car Evaluation features and their corresponding categorical values paired with temperature encodings. Observe that the implied ordering of the feature categories is preserved in the pairwise hamming distance ordering for the encodings. For example, the encoding for the `very high` value of the `buying price` feature is one unit away from `high`, two units away from `medium`, and three units away from `low`. Likewise, this behavior can be observed in the selected encoding of the remaining features in the table. A total input dimension of 15 is required to sufficiently encode the feature space in this manner, imposing less demand on computational resources than one-hot encoding but not quite to the same extent as bipolar encoding. This temperature encoding is expected to better suit the processing unit performance due to the more effective capturing of the categorical rankings in a manner perceivable by the processing unit.

The usual leave-one-out cross-validation experiment is repeated in Table 5.7 on the temperature encoded Car Evaluation data set with entry masking generalization (a) and entry flipping generalization (b). The accuracy ratings are again parameterized by masking level l and exponential weight w. As noted in the previous experiments, no generalization occurs when $l = 0$ so the processing unit guesses the majority class with accuracy equal to the class distribution. The column with masking level $l = 1$ records significant improvement in the processing unit predictive accuracy, with a rating of 97.3958% regardless of the choice of w. This is the maximum observed accuracy for this experiment, handily defeating the maximum accuracies observed with the bipolar and one-hot encoding methods. This supports

Table 5.6: Car Evaluation data features and their respective categorical values, each value paired with a temperature encoding meant to capture the implied value ordering.

Feature	Values
buying price	{very high : $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$}, {high : $\begin{pmatrix} -1 & 1 & 1 \end{pmatrix}$}, {medium : $\begin{pmatrix} -1 & -1 & 1 \end{pmatrix}$}, {low : $\begin{pmatrix} -1 & -1 & -1 \end{pmatrix}$}
maintenance cost	{very high : $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$}, {high : $\begin{pmatrix} -1 & 1 & 1 \end{pmatrix}$}, {medium : $\begin{pmatrix} -1 & -1 & 1 \end{pmatrix}$}, {low : $\begin{pmatrix} -1 & -1 & -1 \end{pmatrix}$}
number of doors	{2 : $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$}, {3 : $\begin{pmatrix} -1 & 1 & 1 \end{pmatrix}$}, {4 : $\begin{pmatrix} -1 & -1 & 1 \end{pmatrix}$}, {5 more : $\begin{pmatrix} -1 & -1 & -1 \end{pmatrix}$}
occupancy	{2 : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {4 : $\begin{pmatrix} -1 & 1 \end{pmatrix}$}, {more : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}
luggage boot size	{small : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {medium : $\begin{pmatrix} -1 & 1 \end{pmatrix}$}, {big : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}
safety	{low : $\begin{pmatrix} 1 & 1 \end{pmatrix}$}, {medium : $\begin{pmatrix} -1 & 1 \end{pmatrix}$}, {high : $\begin{pmatrix} -1 & -1 \end{pmatrix}$}

the notion that the previously considered encoding methods are ill-suited at capturing the inherent relationships between the feature values in a manner detectable by the processing unit architecture. However, notice that the selected values of w are significantly larger in scale than those utilized in the previous experiments. It took very large w for masking levels $l > 1$ to achieve the maximum observed accuracy for both generalization mechanisms. This may be attributed to the encoding method since this behavior was not previously observed. The majority class has substantial representation, comprising 1210 of the 1728 data instances, whereas two other classes are represented by only 69 and 65 data instances. It may be the case that with $l \geq 2$ some target test vectors from these underrepresented classes are close enough in Hamming distance to a substantial amount of training instances

belonging to the majority class, and these vectors are thereby erroneously classified accordingly. Large exponential weight w is required to strip these vectors away from the majority class to be correctly classified. This behavior may not have been detected in the previous experiments since the accuracy ratings were significantly lower and the processing unit may have been misclassifying these edge cases regardless of the hyperparameter settings. Figure 5.5 plots the trend towards maximum observed accuracy as w increases with entry masking (a) and entry flipping (b). Figure 5.6 plots a comparison of performance between the entry masking and entry flipping mechanisms, indicating that both generalization techniques achieve comparable performance when w is sufficiently large.

It is evident that temperature encoding better suits the processing unit performance on the Car Evaluation data set, but the bipolar and one-hot encodings should not be completely discounted. Some data sets may contain categorical features that have no implied ordering, and it would be erroneous to impose some manufactured Hamming distance relations where none are warranted. In such cases, it would be wise to consider one of the earlier encoding techniques, especially one-hot encoding since all of the feature categories will be pairwise equidistant. Some data sets may call for a combination of these encodings depending on the behavior of the features involved. As such, these experimental studies should serve as flexible suggestions rather than rigid commands.

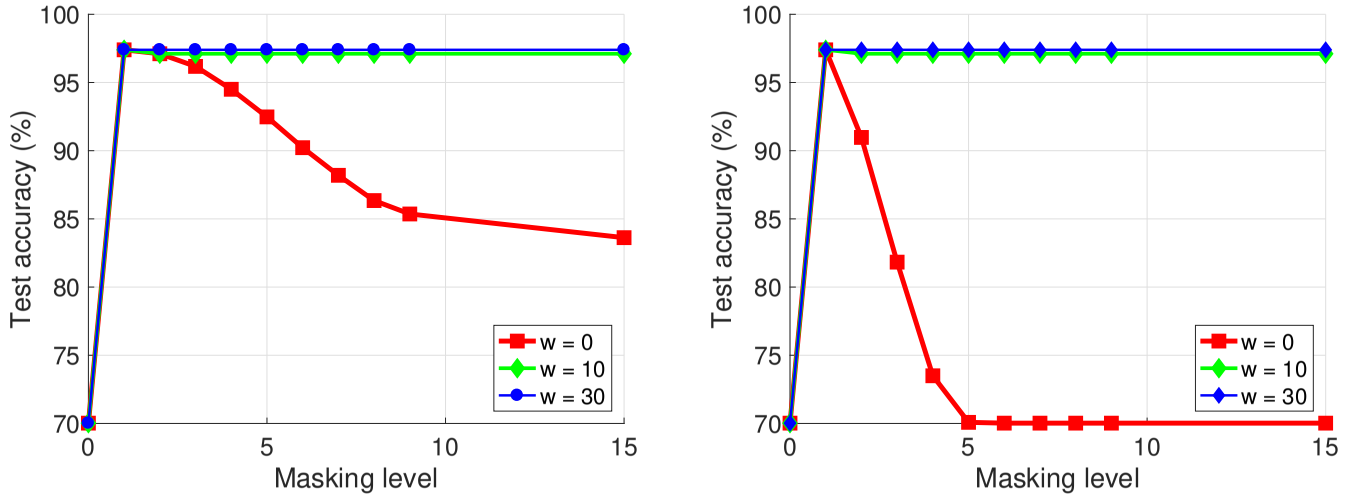
Table 5.7: Leave-one-out cross-validation accuracies on temperature encoded Car Evaluation data by masking level and masking level weight. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

(a) Accuracy ratings with entry masking generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=15$
0	70.0231	97.3958	97.1065	96.1806	94.5023	92.4769	90.2199	88.1944	86.3426	85.3588		83.6227
5	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065		97.1065
10	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065		97.1065
15	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	...	97.1065
20	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065		97.1065
25	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065		97.1065
30	70.0231	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958		97.3958

(b) Accuracy ratings with entry flipping generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=15$
0	70.0231	97.3958	90.9722	81.8287	73.4954	70.0810	70.0231	70.0231	70.0231	70.0231		70.0231
5	70.0231	97.3958	97.1065	97.1644	97.1644	97.1644	97.1644	97.1644	97.1644	97.1644		97.1644
10	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065		97.1065
15	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	...	97.1065
20	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065		97.1065
25	70.0231	97.3958	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065	97.1065		97.1065
30	70.0231	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958	97.3958		97.3958



(a) Entry masking

(b) Entry flipping

Figure 5.5: Learning curves of a processing unit applied to temperature encoded Car Evaluation data by masking level. Accuracy ratings are shown for quantities $w = 0$, $w = 10$, and $w = 30$ with entry masking generalization (a) and entry flipping generalization (b). Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

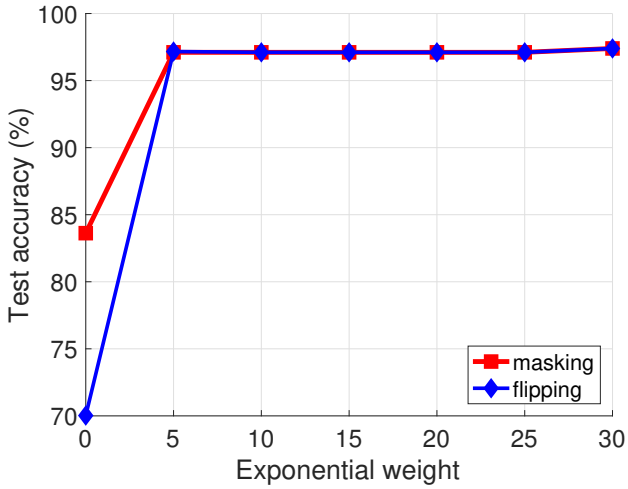


Figure 5.6: Learning curves of a processing unit applied to temperature encoded Car Evaluation data by exponential weight w with masking level $l = 15$. Accuracy ratings are shown for entry masking generalization and entry flipping generalization. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

5.2.4 Comparison with Historical Results

In this experiment, 10-fold cross-validation is used to produce a sample mean accuracy rating with which to compare to known historical results reported in other publications. Several well-known learning algorithms have been applied to the Car Evaluation data set and reported in other publications for comparison, analysis, and development. Using 10 times repeated 10-fold cross-validation, the C4.5 and C5 decision tree algorithms are reported to achieve accuracy ratings of $92.2 \pm 2.2\%$ and $92.2 \pm 2.1\%$, respectively [23]. The Waikato Environment for Knowledge Analysis (WEKA) [3, 17] implementation of the Naive Bayes classifier is reported to achieve

85.706% 10-fold cross-validation accuracy [20]. The same publication also reports a 10-fold cross-validation accuracy rating of 99.537% using the WEKA implementation of the multilayer perceptron.

With the execution of 10-fold cross-validation, stratified sampling is used to partition the 1728 Car Evaluation data instances into 10 subsets of roughly equal size. The processing unit is trained on 9 of the 10 subsets and tested on the remaining subset. This procedure is iterated 10 times so that each subset serves as a test set. The 10 test set accuracies are then averaged to produce a sample 10-fold cross-validation accuracy rating. This sample accuracy is dependent on the partitioning of the data, so the 10-fold cross-validation procedure is repeated 1,000 times in order to produce a more stable overall mean accuracy rating. Stratified sampling is used to ensure that the partitions contain roughly the same class distributions.

The processing unit is configured according to the prior experiments examining the influence of the hyperparameters on the observed leave-one-out cross-validation accuracy ratings. The maximum reported leave-one-out cross-validation accuracy is 97.3958% with the temperature encodings and masking matrix configuration discussed in Section 5.2.3. This is achieved with multiple values of masking level l and exponential weight w, but $l = 15$ and $w = 30$ are selected for this experiment. Figure 5.7 depicts a histogram plot of the 1,000 observed 10-fold cross-validation accuracy ratings. The histogram bars have width 0.1 and height proportional to the frequency of the accuracy ratings. Table 5.8 summarizes some statistical measures reported in this experiment.

It is evident that the processing unit outperforms the aforementioned historical

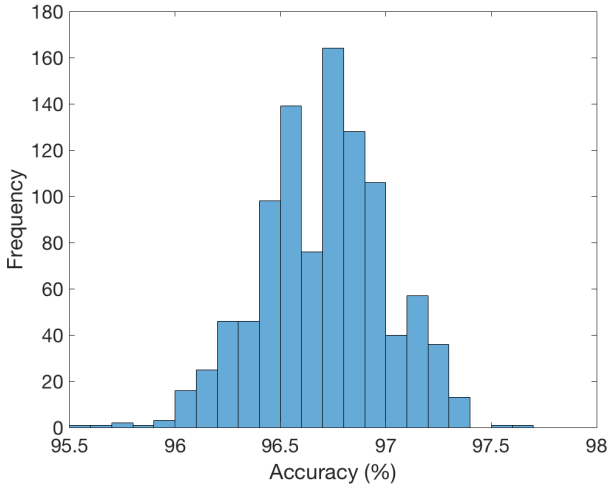


Figure 5.7: Histogram plot of the 1,000 stratified 10-fold cross-validation accuracy ratings of the THPAM processing unit applied to the Car Evaluation data set. The bar heights are proportional to the frequency of the observed accuracies, and the bar widths are 0.1. The input encoding scheme discussed in Section 5.2.3 is used, and masking generalization is utilized with $l = 15$ and $w = 30$.

results pertaining to the C4.5 and C5 decision trees and the Naive Bayes classifier, but the reported multilayer perceptron accuracy of 99.537% remains out of reach. This result is reproducible with WEKA version 3.8.1, which automatically constructs a multilayer perceptron consisting of 21 input nodes, 12 hidden nodes, and 4 output nodes when applied to the Car Evaluation data set. The 21 input nodes correspond with the 21 total input feature values, and the 4 output nodes correspond with the 4 class labels. On a machine running macOS version 10.12.4 with 16GB 1866 MHz PC3-14900 LPDDR3 memory and a 2.0 GHz dual-core Intel Core i5 Skylake (6360U) with 4 MB L3 cache, the default WEKA version 3.8.1 implementation of the multilayer perceptron performs 500 iterations to learn the entire training set over 4.33 seconds. A serial implementation of THPAM in C running on the same

Table 5.8: Summary statistics of the 1,000 stratified 10-fold cross-validation accuracy ratings of the THPAM processing unit applied to the Car Evaluation data set. The input encoding scheme discussed in Section 5.2.3 is used, and masking generalization is utilized with $l = 15$ and $w = 30$.

Statistic	Accuracy (%)
Mean	96.7023 ± 0.2975
Maximum	97.6273
Minimum	95.5440
Mode	96.7014

hardware requires only a single iteration to learn the data set over 0.07 seconds. It should be noted that the default 10-fold cross-validation procedure in WEKA is non-repeating and utilizes a default seed to partition the data unless otherwise specified. Thus, there remains some discrepancy between the historical results and the processing unit performance herein, though the historical results cannot be completely discounted.

Although the present realization of the processing unit does not achieve a performance result comparable to that of the multilayer perceptron, there is a strong possibility that the processing unit can achieve such a performance with further research development. In Section 5.4 it is demonstrated that although the processing unit cannot currently detect the relative significance of features when performing classification, it is possible to manually implement feature weights within the construction of the masking matrix in order to improve performance. An automatic method of manufacturing these feature weights within the masking matrix is desirable and promising.

5.3 UCI Congressional Voting Records Data Set

The UCI Congressional Voting Records data set [21] consists of binary voting histories of elected representatives in the 98th US congress on key issues identified by the Congressional Quarterly Almanac. There are 16 features, each of which identify whether the corresponding representative voted yay or nay on the respective bill. Although the features are binary, there is technically a third category denoting an unknown position due to abstention. The data set is not exhaustive, and there are some repeated data instances. The voting histories are used to predict the party membership of each representative, whether they be `democrat` or `republican`. Table 5.9 lists these classes and their distributions among the instance data. Of the 435 data instances, the majority class is `democrat` with 61.3793% representation, which should be the lower bound on the accuracy of a learning algorithm on this data set since a rudimentary algorithm could blindly guess this class.

Table 5.9: Congressional Voting data set classes and their respective distributions among the instance data.

Class	# of instances	% of instances
democrat	267	61.3793%
republican	168	38.6207%
total	435	

We again examine the performance of the THPAM processing unit on this data set with different encoding methods. The data is not provided with distinct training and testing sets. Leave-one-out cross-validation is again selected as the performance measure used to evaluate the processing unit under the different encoding methods.

Class labels are represented using one-hot bipolar vectors. Class predictions are generated by producing the empirical probability then selecting the class associated with the entry that has greatest probability relative to the other entries. Any ties are broken by selecting the class which has greater representation in the instance data.

5.3.1 Temperature Encoding

In this experiment we treat the unknown or missing votes as a third class, making the data set features ternary rather than binary. In encoding these features, we utilize the temperature encoding procedure previously determined to be the most suitable representation of the Car Evaluation data set for the processing unit performance. Table 5.10 lists suitable encodings that were selected for each of the feature values. In this arrangement, it is assumed that any `unknown` values should be equidistant in Hamming distance from both the `yay` and `nay` values since abstention is implied to be not a vote for nor a vote against anything. The distance from `unknown` to the other two values is thereby constructed to be 1. The distance between values `yay` and `nay` should therefore be greater than 1, so these encodings are constructed to be a distance of 2 from each other. All of the 16 features are encoded in this same manner, so the bipolar inputs will be of dimension 32. A single computer processor is certainly too limited to perform this experiment with a processing unit on this input size, and this is even brushing against the user limitations on the UMBC HPCF parallel computing cluster. Nevertheless, some

results are obtainable with some patience and opportunity.

Table 5.10: Congressional Voting data features and their respective categorical values, each value paired with a temperature encoding meant to capture the implied value ordering.

Value	Encoding
yay	$\begin{pmatrix} 1 & 1 \end{pmatrix}$
nay	$\begin{pmatrix} -1 & -1 \end{pmatrix}$
unknown	$\begin{pmatrix} 1 & -1 \end{pmatrix}$

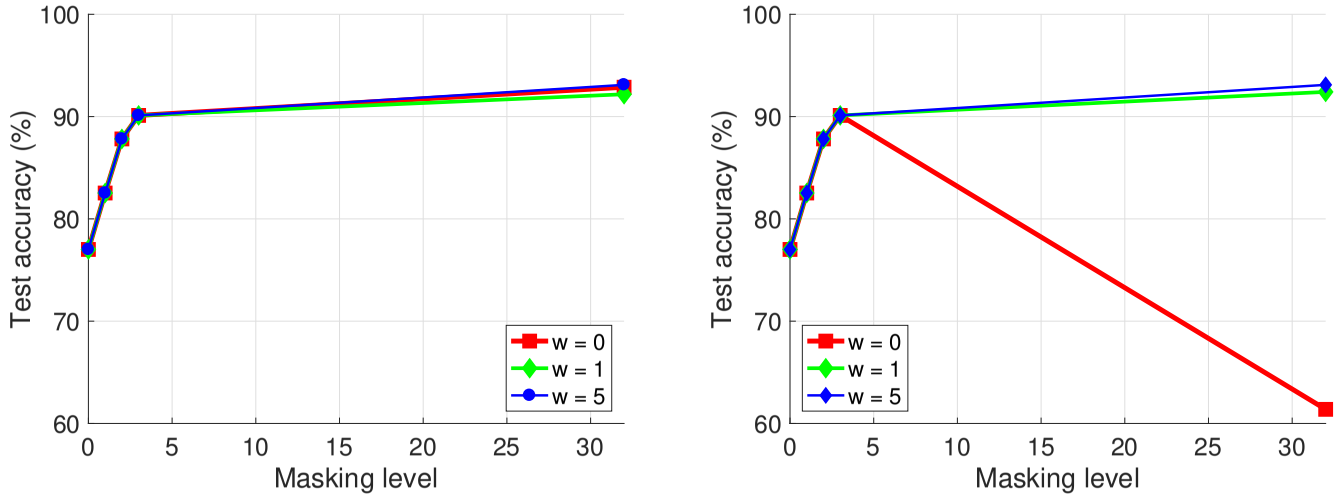
Table 5.11 lists the observed leave-one-out cross-validation accuracies on the temperature encoded Congressional Voting data set with entry masking (a) and entry flipping (b). The accuracy ratings are parameterized by masking level l and exponential weight w. Due to the high dimensionality of the encoded inputs, only a few select values of l were involved in this study, namely $0 \leq l \leq 3$ and $l = 32$. Masking level $l = 32$ is achievable with the alternative method of masking matrix construction introduced in Section 3.3. Additional masking levels could not be used in this study due to the time complexity of the general masking matrix formula, requiring more parallel computing resources than users are allotted on UMBC HPCF. Regardless, these few results are useful for comparison purposes with the second encoding method discussed later.

Recall the earlier note that the data set contains some repeated data instances. As such, the predictive accuracy without generalization, i.e. $l = 0$, is reported as 77.0115% rather than the 61.3793% corresponding with the majority class distribution. With so few masking level values considered, it is difficult to identify which value of l would produce an optimal accuracy other than $l = n = 32$. Based on pre-

vious experiments it may be reasonable to assume that an optimal or near optimal accuracy is attainable with $l = n$ providing w is suitably selected. The maximum accuracy observed in this experiment is 93.1034%, which should be optimal or near optimal since this performance remains stable with $l = 32$ and increasing w. It is apparent that adjustments in the exponential weight have little influence on the performance for this data set, except when $l = 32$ and likely other masking levels which were not considered. This can be attributed to the fact that the accuracies are ever increasing with the reported masking level values, so an optimal value of l after which these accuracies begin to decrease was not identified. The role of the exponential weight is to prevent such a decrease, but this has no impact on the lower masking level values at which the accuracy could be improved by raising the masking level. This is true only until w is selected to be exceedingly large, at which point the lower level entry maskings will have more relative weight than the higher level entry maskings, regardless of the choice of l. Still, the exponential weight remains useful for optimizing the performance with $l = 32$. Figure 5.8 plots the test accuracies by the masking levels for selected exponential weights $w = 0$, $w = 1$, and $w = 5$ with entry masking (a) and entry flipping (b). The trend towards the maximum observed accuracy can be identified only with masking level $l = 32$. Figure 5.9 plots this trend in a comparison between entry masking and entry flipping performance when $l = 32$, which indicates that both generalization mechanisms achieve similar performance when w is sufficiently large.

Table 5.11: Leave-one-out cross-validation accuracies on temperature encoded Congressional Voting data by masking level and masking level weight. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

(a) Accuracy ratings with entry masking generalization						
w	$l = 0$	$l = 1$	$l = 2$	$l = 3$	...	$l = 32$
0	77.0115	82.5287	87.8161	90.1149		92.8736
1	77.0115	82.5287	87.8161	90.1149		92.1839
2	77.0115	82.5287	87.8161	90.1149	...	93.1034
3	77.0115	82.5287	87.8161	90.1149		93.1034
4	77.0115	82.5287	87.8161	90.1149		93.1034
5	77.0115	82.5287	87.8161	90.1149		93.1034
(b) Accuracy ratings with entry flipping generalization						
w	$l = 0$	$l = 1$	$l = 2$	$l = 3$	...	$l = 32$
0	77.0115	82.5287	87.8161	90.1149		61.3793
1	77.0115	82.5287	87.8161	90.1149		92.4138
2	77.0115	82.5287	87.8161	90.1149	...	92.6437
3	77.0115	82.5287	87.8161	90.1149		93.1034
4	77.0115	82.5287	87.8161	90.1149		93.1034
5	77.0115	82.5287	87.8161	90.1149		93.1034



(a) Entry masking

(b) Entry flipping

Figure 5.8: Learning curves of a processing unit applied to temperature encoded Congressional Voting data by masking level. Accuracy ratings are shown for quantities $w = 0$, $w = 1$, and $w = 5$ with entry masking generalization (a) and entry flipping generalization (b). Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

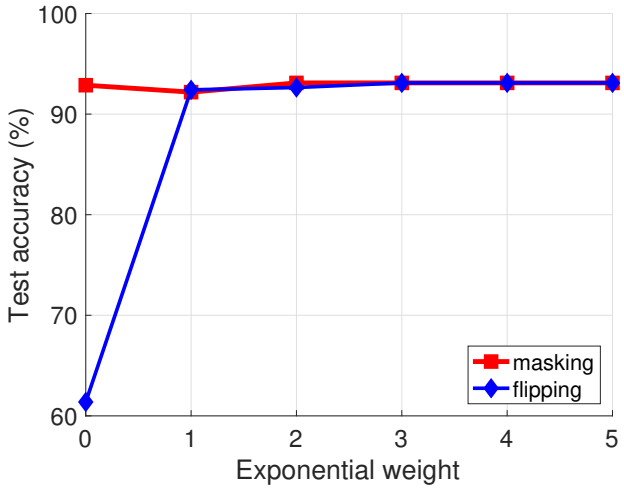


Figure 5.9: Learning curves of a processing unit applied to temperature encoded Congressional Voting data by exponential weight w with masking level $l = 32$. Accuracy ratings are shown for entry masking generalization and entry flipping generalization. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

5.3.2 Ternary Encoding

In this experiment, we examine an alternative method of handling unknown or missing data rather than treating it as a separate class equidistant from the other classes. Begin with the assumption that the presented voting features are strictly the binary values of `yay` or `nay`, each of which can be encoded with a single bipolar entry of 1 for `yay` and -1 for `nay`. A missing entry could be either `yay` or `nay` in equal measure, so one option for handling the unknown entry is to learn two distinct data instances, one with $+1$ as the missing entry and the other with -1. However, this becomes untenable when considering that some instances contain many missing

values. A training instance containing k missing values would require the training of 2^k different vectors to accommodate the missing values in this manner. Additionally, this poses a problem when producing the empirical probability on a target vector containing missing values. It is challenging to conceive of a method that would generate fair predictions in this scenario.

A second option that accomplishes the above ideas without exponential blowup is to relax the bipolar constraint on the input vectors. The THPAM processing unit extracts meaning from the values 1 and -1, but technically 0 has a particular meaning as well. Numeral 0 plays a vital role in entry masking, wherein a replacement of zero means that the entry value should be ignored or hidden in order to obtain more information using the other unmasked entries. This procedure of hiding certain entries in favor of others can be extended to the original input vectors when particular entries are missing or unknown. If a feature entry is missing, then it would be preferable to ignore that feature in an attempt to produce an accurate prediction based on the other known feature entries. This must occur in the input vector encodings rather than in entry masking generalization because the masking matrices cannot be used to detect whether or not a specific feature is missing. Thus for this experiment, we encode the data set instances by using 1 to denote `yay`, -1 to denote `nay`, and 0 to denote an unknown value. This results in an input dimension of 16, which is a reduction by half in comparison to the 32-dimensional encodings used the previous experiment. This input dimension is far more reasonable for testing the processing unit program on a serial machine.

Table 5.12 lists the results of the leave-one-out cross-validation study with

varying masking level l and exponential weight w. Table 5.12 (a) shows results of using entry masking generalization. When $w = 0$ the accuracy reaches a maximum of 93.5632% with $l = 5$, which gradually diminishes as l increases further. The results change slightly as w is adjusted. With $w = 1$ and $w = 2$ a slight decline in accuracy can be observed for most values of l, except notably with $l = 2$ which increases towards a maximum accuracy of 93.5632% at $w = 3$. With $w = 3$ and $w = 4$ the accuracy ratings generally increase towards the overall maximum recorded accuracy of 93.7931% for $l \geq 4$. This accuracy is achievable for most values of l when $w = 4$, suggesting that this value is optimal of the values considered. As w increases further, the accuracies decline well below the maximum observed value. This is likely because a masking level of at least $l = 4$ or $l = 5$ is optimal for this data set, but the increase in w raises the weights of the lower level entry maskings relative to the higher level entry maskings. As such, w must not only be sufficiently large but also carefully selected to avoid this overfitting of the training data. Masking up to the maximum masking level of 16 entries is also capable of achieving the optimal accuracy at $w = 4$, demonstrating that masking up to all the input entries is appropriate providing w is suitably selected.

As expected based on the previous experiments, the results of entry flipping generalization in Table 5.12 (b) are generally worse than those of entry masking, since entry flipping does not inherently favor lower masking levels without w being appropriately chosen. However, both generalization schemes tend toward the same maximum accuracy rating of 93.7931%, but this occurs at a slightly larger $w = 5$ in comparison to the optimal accuracy achieved with entry masking at $w = 4$.

Like with entry masking, choosing w to be too large causes a decrease in accuracy from the maximum, since this adjustment increases the relative weights of the lower level entry maskings which are not optimal for this data set. Figure 5.10 plots the accuracies by the masking level for selected values of w with entry masking (a) and entry flipping (b). The benefit of carefully adjusting w can be more clearly observed, as each trend line gradually stabilizes toward optimal accuracy with increasing w up to a point. The choice of exponential weight is especially important with entry flipping since the increase in masking level causes a trend towards the distribution of the majority class when $w = 0$. Nevertheless, the entry masking and entry flipping generalization methods are capable of performing approximately on par with each other if w is made large enough, as evidenced in Figure 5.11.

The processing unit architecture achieves overall better performance with this ternary encoding method compared to the earlier examined temperature encoding method in which missing values were treated as a separate class. The maximum recorded accuracy of 93.7931% is slightly better than the maximum 93.1034% observed in the preceding experiment. Of substantial note is that this greater performance is achieved while utilizing far fewer input entries, making this encoding method much more favorable in both respects of performance and efficiency. This experiment should serve as a suggestion to relax the bipolar restriction on inputs whenever missing data may be anticipated.

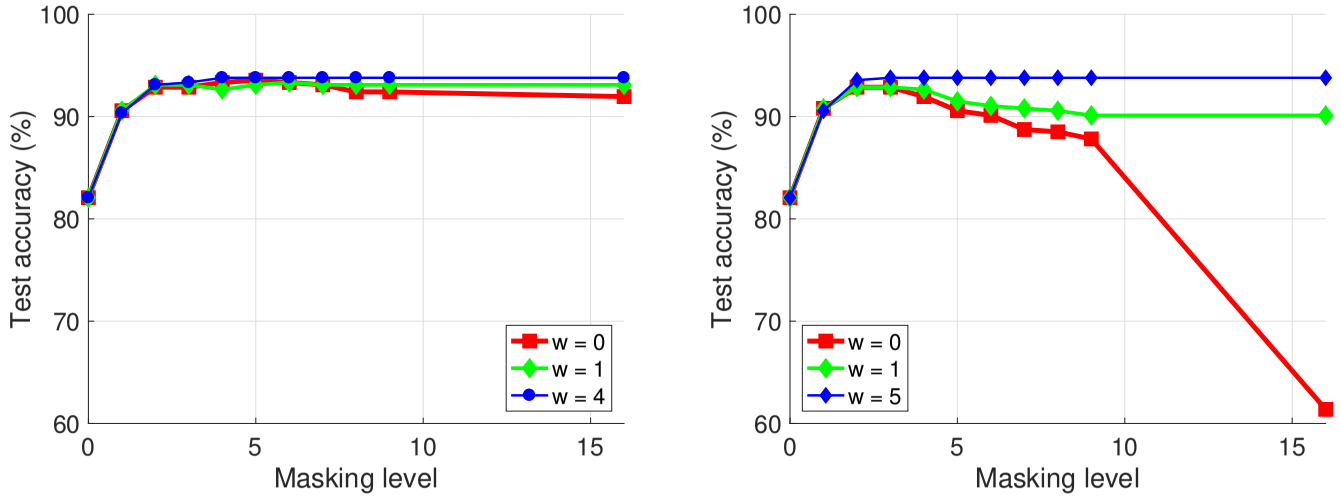
Table 5.12: Leave-one-out cross-validation accuracies on ternary encoded Congressional Voting data by masking level and masking level weight. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

(a) Accuracy ratings with entry masking generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=16$
0	82.0690	90.5747	92.8736	92.8736	93.3333	93.5632	93.3333	93.1034	92.4138	92.4138		91.9540
1	82.0690	90.5747	93.1034	93.1034	92.6437	93.1034	93.3333	93.1034	93.1034	93.1034		93.1034
2	82.0690	90.5747	93.3333	93.1034	92.8736	92.6437	92.8736	92.8736	92.8736	92.8736		92.8736
3	82.0690	90.5747	93.5632	93.1034	93.3333	93.1034	93.1034	93.1034	93.1034	93.7931	...	93.1034
4	82.0690	90.3448	93.1034	93.3333	93.7931	93.7931	93.7931	93.7931	93.7931	93.7931		93.7931
5	82.0690	90.3448	92.4138	92.8736	92.8736	92.8736	92.8736	92.8736	92.8736	92.8736		92.8736
6	82.0690	90.1149	91.4943	91.9540	91.9540	91.9540	91.9540	91.9540	91.9540	91.9540		91.9540

(b) Accuracy ratings with entry flipping generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=16$
0	82.0690	90.8046	92.8736	92.8736	91.9540	90.5747	90.1149	88.7356	88.5057	87.8161		61.3793
1	82.0690	90.8046	92.8736	92.8736	92.6437	91.4943	91.0345	90.8046	90.5747	90.1149		90.1149
2	82.0690	90.5747	92.6437	93.3333	93.3333	93.5632	93.1034	92.8736	93.1034	93.1034		93.1034
3	82.0690	90.5747	92.8736	92.8736	93.1034	93.1034	93.1034	93.1034	93.1034	93.1034	...	93.1034
4	82.0690	90.5747	92.8736	93.1034	93.1034	93.1034	93.1034	93.1034	93.1034	93.1034		93.1034
5	82.0690	90.5747	93.5632	93.7931	93.7931	93.7931	93.7931	93.7931	93.7931	93.7931		93.7931
6	82.0690	90.3448	92.6437	92.8736	92.8736	92.8736	92.8736	92.8736	92.8736	92.8736		92.8736



(a) Entry masking

(b) Entry flipping

Figure 5.10: Learning curves of a processing unit applied to ternary encoded Congressional Voting data by masking level. Accuracy ratings are shown for quantities $w = 0$, $w = 1$, and $w = 4$ with entry masking generalization (a) and $w = 0$, $w = 1$, and $w = 5$ with entry flipping generalization (b). Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

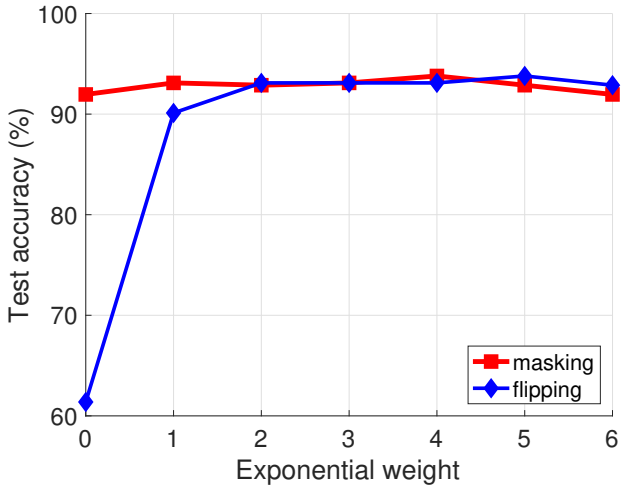


Figure 5.11: Learning curves of a processing unit applied to ternary encoded Congressional Voting data by exponential weight w with masking level $l = 16$. Accuracy ratings are shown for entry masking generalization and entry flipping generalization. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

5.3.3 Comparison with Historical Results

In this experiment, 10-fold cross-validation is again used to produce a sample mean accuracy rating with which to compare to the performance of other known learning algorithms reported in other work. With 10 times repeated 10-fold cross-validation, the WEKA implementations of the J48 decision tree and Naive Bayes algorithms are reported to achieve mean accuracies of 96.46±0.17% and 90.18±0.07%, respectively [1]. The default WEKA version 3.8.1 implementation of the multilayer perceptron achieves a 10-fold cross-validation accuracy of 94.71%, configured with 16 input nodes, 9 hidden nodes, and 2 output nodes. This result is obtained with

the default seed value for partitioning the data set into 10 stratified folds.

The processing unit is configured according to the prior experiments examining the influence of the hyperparameters on the observed leave-one-out cross-validation accuracy ratings. The maximum reported leave-one-out cross-validation accuracy is 93.7931% with the ternary input encodings and masking matrix configuration discussed in Section 5.3.2. This is achieved with multiple values of masking level l and exponential weight w, but $l = 16$ and $w = 4$ are selected for this experiment. Figure 5.12 depicts a histogram plot of the 1,000 observed 10-fold cross-validation accuracy ratings. The histogram bars have width 0.25 and height proportional to the frequency of the accuracy ratings. Table 5.13 summarizes some statistical measures reported in this experiment.

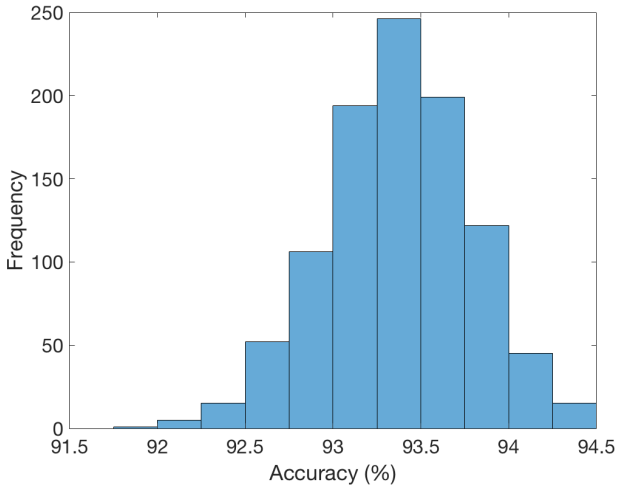


Figure 5.12: Histogram plot of the 1,000 stratified 10-fold cross-validation accuracy ratings of the THPAM processing unit applied to the Congressional Voting data set. The bar heights are proportional to the frequency of the observed accuracies, and the bar widths are 0.25. The input encoding scheme discussed in Section 5.3.2 is used, and masking generalization is utilized with $l = 16$ and $w = 4$.

Table 5.13: Summary statistics of the 1,000 stratified 10-fold cross-validation accuracy ratings of the THPAM processing unit applied to the Congressional Voting data set. The input encoding scheme discussed in Section 5.3.2 is used, and masking generalization is utilized with $l = 16$ and $w = 4$.

Statistic	Accuracy (%)
Mean	93.3301 ± 0.3868
Maximum	94.4828
Minimum	91.9540
Mode	93.3333

The processing unit is capable of outperforming the Naive Bayes algorithm, and the best reported performance of 94.4828% approaches that of the multilayer perceptron performance. On a machine running macOS version 10.12.4 with 16GB 1866 MHz PC3-14900 LPDDR3 memory and a 2.0 GHz dual-core Intel Core i5 Skylake (6360U) with 4 MB L3 cache, the default WEKA version 3.8.1 implementation of the multilayer perceptron performs 500 iterations to learn the entire training set over 0.68 seconds. A serial implementation of THPAM in C running on the same hardware requires only a single iteration to learn the data set over 0.03 seconds. Similar to the Car Evaluation data set performance, there is opportunity for improvement and the results are promising in these initial benchmark studies. The generalization capability of the processing unit can be improved by devising an automatic method of selecting masking matrix weights according to the relative significance of the input features in determining the class labels. The performance of the J48 decision tree algorithm shows that such a mechanism would be critical in the performance on this data set. This decision tree is constructed according to the relative information gain of splitting the data set based on the feature values, giving

precedence to features which are most influential in determining the class labels. Such a procedure is manually demonstrated to be possible via the masking matrix weights as discussed in the next section pertaining to the UCI Iris data set.

5.4 UCI Iris Data Set

The UCI Iris data set [5] consists of real-valued measurements regarding select characteristics of iris plants. These characteristics include sepal length, sepal width, petal length, and petal width, all provided in centimeters, which are used in classification of the iris plant types. There are three classes, namely `iris setosa`, `iris versicolour`, and `iris virginica`, all of which are distributed equally among the 150 data set instances. Therefore, there is no majority class and the lower bound on predictive accuracy is 33.3333% with random guessing. The presence of real-valued data poses a challenge for applying the THPAM processing unit since real-valued data is not as conveniently represented in a binary or bipolar encoding. Nevertheless, an attempt at this is made in this work, and though the method may be unconventional, there is a novel way of constructing the entry masking and flipping matrices to better suit the circumstance of this experiment.

Table 5.14 lists each of the data set features, their minimum and maximum values, and the number of values that are possible for each feature. Although the data is real-valued, only two significant digits are reported and the range of potential values is not substantial. This is advantageous in producing an input encoding suitable for the processing unit. For each feature, we rank the values in increasing

Table 5.14: Iris data features, their minimum and maximum values, and the number of possible feature values.

Feature	Minimum value	Maximum value	Number of values
sepal length	4.3	7.9	37
sepal width	2.0	4.4	25
petal length	1.0	6.9	60
petal width	0.1	2.5	25

order. For example, the minimum value of 4.3 for the `sepal length` feature would have rank 0, and the maximum value of 7.9 would have rank 36. The assigned numeric ranks are then encoded into their standard binary representation, except zeros are replaced with negatives to produce bipolar encoded vectors. These bipolar encodings increase in significance from left to right, where the leftmost entry represents $2^0 = 1$ and the rightmost entry represents $2^4 = 16$ or $2^5 = 32$ depending on the number of ranks required to entirely represent the feature. Given the number of possible values of each feature listed in Table 5.14, the `sepal length` and `petal length` features require 6 bits of representation, and the `sepal width` and `petal width` features require 5 bits of representation. Thus, a total bipolar input size of 22 entries is necessary to encode the data set instances in this manner. This input size is too large for a serial processing unit program to process within a reasonable amount of time, but the execution of the parallelized implementation of the processing unit on a parallel computing cluster suffices to overcome this burden with relative ease.

Similar to the previously examined data sets, the Iris data set lacks distinct training and testing sets, so leave-one-out cross-validation is selected to measure

the performance of the processing unit. The three class labels are represented with one-hot encoded bipolar vectors, and class predictions are produced by constructing the empirical probability then selecting the class associated with the entry that has greatest probability relative to the other entries. Any ties are broken by choosing a class in the following order: first `iris virginica`, second `iris versicolour`, and third `iris setosa`. Since there is no majority class, breaking ties in this order may not be better than any other order, but having any such order is beneficial for producing consistent results. The subsequent subsections each examine the performance of the processing unit on the aforementioned bipolar encoding of the Iris data set, with a distinct difference to be discussed.

5.4.1 Bipolar Encoding

In this experiment, the processing unit performance is measured on the bipolar encoding of the data instances in the usual manner. Table 5.15 lists the results of this experiment with varying exponential weight w and masking level l with entry masking (a) and entry flipping (b) generalization methods. The results reveal some unusual behaviors pertaining to the adjustments of the hyperparameters. Overall, it appears that increasing w has little impact on the performance, and sometimes the performance declines as observed with masking level $l = 22$. The performance appears to be best when $w = 0$, removing the masking level weights entirely. This is especially notable with entry flipping generalization (b) which has no inherent favoring of lower masking levels. The maximum observed accuracy of 93.3333% is

achieved only with entry flipping generalization with $l = 6$ and $l = 7$ with very little or no exponential weight. Masking matrix generalization fails to achieve the same maximum accuracy with the hyperparameter values considered in this experiment. Figure 5.13 plots trend lines of the test accuracy over the masking level for some select values of w with entry masking (a) and entry flipping (b). The plots more clearly show that nonzero w appears to do more harm than good, except of course with the entry flipping matrix when l is fairly large. Figure 5.14 compares the performance of entry masking with entry flipping when $l = 22$ with varying exponential weight, showing that with both methods achieve similar performance when w is sufficiently large, even if this parameter may be detrimental for this data set encoding.

The unusual findings in this experiment may be attributed to the encoding method used. In machine learning and data analysis, it is generally expected that some data features may have more significance than others in data classification. This is clearly true for the bipolar encoding used herein, since the bits in the binary representation of numbers do not have equal significance. The bit associated with $2^0 = 1$ has substantially less importance than the bit associated with $2^5 = 32$, and the bits in between certainly reveal similar disproportions. However, the processing unit performs generalization in an entirely neutral manner without favoring particular input entries over others. At the present stage in the theoretical development of the processing unit, it is not yet equipped with an automatic method of detecting the significance of input features as they relate to the output. If such a method could be conceived, the masking matrix may be constructed in a manner which reduces

the masking level weight for entry maskings which hide these significant features in order to discourage such masking. The next experiment examines a manual usage of these masking entry weights which leads to some improvement in the processing unit performance.

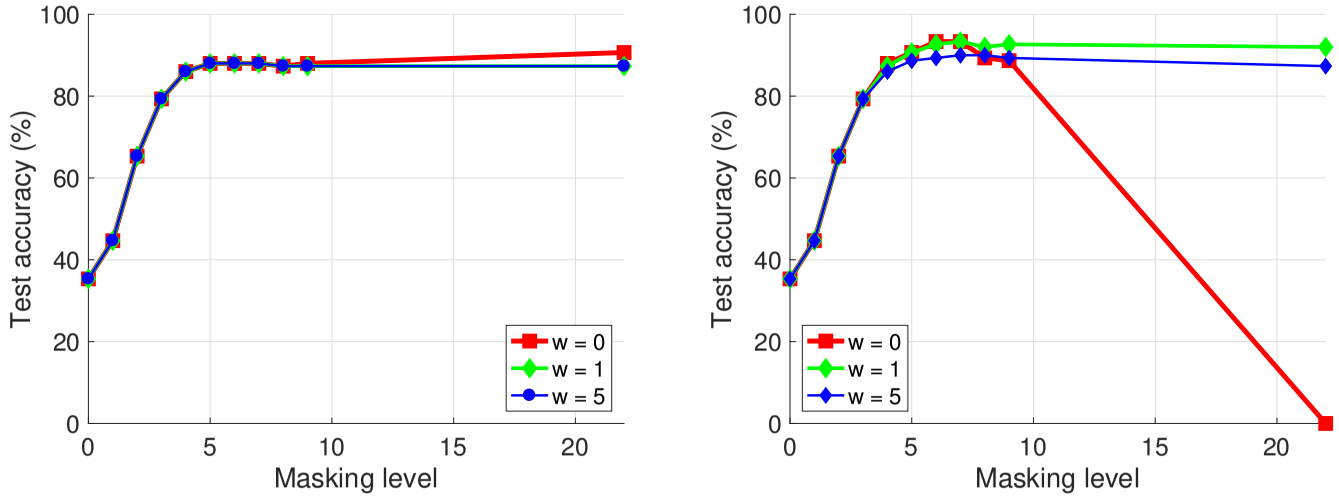
Table 5.15: Leave-one-out cross-validation accuracies on bipolar encoded Iris data by masking level and masking level weight. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

(a) Accuracy ratings with entry masking generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=22$
0	35.3333	44.6667	65.3333	79.3333	86.0000	88.0000	88.0000	88.0000	87.3333	88.0000		90.6667
1	35.3333	44.6667	65.3333	79.3333	86.0000	88.0000	88.0000	88.0000	87.3333	87.3333		87.3333
2	35.3333	44.6667	65.3333	79.3333	86.0000	88.0000	88.0000	88.0000	87.3333	87.3333	...	87.3333
3	35.3333	44.6667	65.3333	79.3333	86.0000	88.0000	88.0000	88.0000	87.3333	87.3333		87.3333
4	35.3333	44.6667	65.3333	79.3333	86.0000	88.0000	88.0000	88.0000	87.3333	87.3333		87.3333
5	35.3333	44.6667	65.3333	79.3333	86.0000	88.0000	88.0000	88.0000	87.3333	87.3333		87.3333

(b) Accuracy ratings with entry flipping generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=22$
0	35.3333	44.6667	65.3333	79.3333	88.0000	90.6667	93.3333	93.3333	89.3333	88.6667		0.0000
1	35.3333	44.6667	65.3333	79.3333	87.3333	90.6667	92.6667	93.3333	92.0000	92.6667		92.0000
2	35.3333	44.6667	65.3333	79.3333	86.0000	88.6667	89.3333	90.0000	90.0000	89.3333	...	89.3333
3	35.3333	44.6667	65.3333	79.3333	86.0000	88.6667	89.3333	90.0000	90.0000	89.3333		87.3333
4	35.3333	44.6667	65.3333	79.3333	86.0000	88.6667	89.3333	90.0000	90.0000	89.3333		87.3333
5	35.3333	44.6667	65.3333	79.3333	86.0000	88.6667	89.3333	90.0000	90.0000	89.3333		87.3333



(a) Entry masking

(b) Entry flipping

Figure 5.13: Learning curves of a processing unit applied to bipolar encoded Iris data by masking level. Accuracy ratings are shown for quantities $w = 0$, $w = 1$, and $w = 5$ with entry masking generalization (a) and entry flipping generalization (b). Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

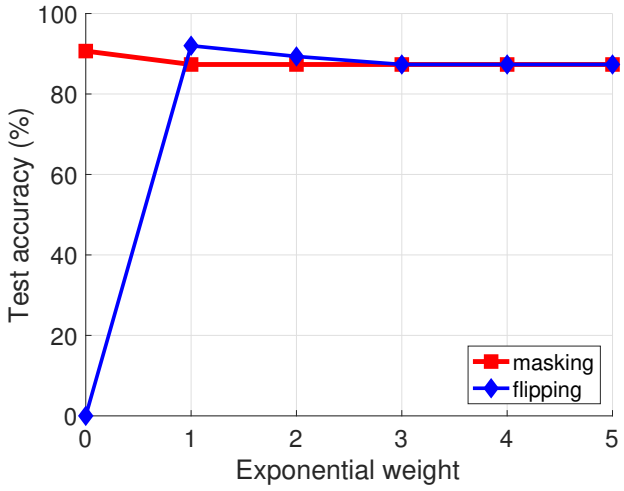


Figure 5.14: Learning curves of a processing unit applied to bipolar encoded Iris data by exponential weight w with masking level $l = 22$. Accuracy ratings are shown for entry masking generalization and entry flipping generalization. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

5.4.2 Bipolar Encoding with Custom Masking Weights

In this experiment, the processing unit learns the same bipolar encoding representations of the Iris data, but the generalization mechanisms are tweaked to better account for the disproportionate significance of the encoding entries. Each of the four features are encoded in standard binary notation, but with -1 in place of 0 to create bipolar vectors. There are 5 or 6 bipolar bits per feature, and each successive bit doubles in the significance compared to the previous bit. The generalization mechanisms are unable to detect this, so the masking of a bits representing $2^0 = 1$ and $2^5 = 32$ have equal weight. A more reasonable arrangement would be to give

substantially less weight to the bit entry for 32 than the bit entry for 1.

Table 5.16 lists a weighting scheme for up to 6 bits of this bipolar representation [16]. After some manual trial and error, this weighting method appeared to best counterbalance the disproportionate significance of the bipolar encoding bits. To summarize, the two leftmost bits of each feature denote values of 1 and 2, which are assigned entry masking weight 1 since these are not very substantial compared with the subsequent bits. The bit associated with value 4 is assigned entry masking weight 2^{-w} where w is an adjustable exponential weight. Each subsequent bit is assigned entry masking weight proportional to the square of the previous masking weight, which reflects the general masking level weights seen in previous experiments. This weighting method is applied in the construction of the masking and flipping matrices, where these weights are applied to their respective single bit entry maskings. If an entry masking masks more than one bit, then the associated weight is the multiple of the single bit entry masking weights. For example, if the bits of entries representing values 16 and 32 are both masked, then that entry masking will receive weight $2^{-3w}2^{-4w} = 2^{-7w}$. Each of the four encoded features are masked according to this custom weighting method.

Table 5.16: Iris data encoding values and associated entry masking weights.

Entry value	1	2	4	8	16	32
Weights	1	1	2^{-w}	2^{-2w}	2^{-3w}	2^{-4w}

Table 5.17 records the leave-one-out cross-validation results of the processing unit applied to the bipolar encoded Iris data set with the custom masking weights

for entry masking (a) and entry flipping (b). These results should be compared to the previous experiment in which the standard masking level weights are utilized. Comparable performance can be seen when $l \leq 3$, but greatly improved generalization performance can be observed as l increases. Overall, the accuracies observed in this experiment reflect the expected relationship between l and w that was demonstrated in experiments on other data sets. Specifically, l is optimized to achieve the best performance, but any further increasing of l can be counterbalanced by also increasing w. This behavior is plotted in Figure 5.15 for entry masking (a) and entry flipping (b) with $w = 0, 1$, and 5. It is notable that the processing unit is capable of achieving the maximum observed accuracy of 96.6667% with entry masking, but this same result could not be attained with entry flipping. As usual, both generalization methods perform approximately the same when w is sufficiently large, as indicated in Figure 5.16 with $l = 22$. However, entry masking appears to maintain a slight advance over entry flipping, since the maximum observed accuracy with entry flipping is 95.3333%. Although not reported here, further increasing w beyond 5 did not remove this difference in comparative performance. The reason for this difference is unclear, but it may be that the masking matrix better suits the custom masking level weights selected for this experiment. In handcrafting these masking level weights, the objective was to maximize the processing unit accuracy while using entry masking rather than entry flipping. The same masking level weights were then applied in the construction of the flipping matrix without consideration that another set of custom weights might be more suitable for the performance of entry flipping.

The Iris data set experiments demonstrate an important finding regarding the processing unit generalization methods. These methods are unable to detect feature significance and adjust the masking level weights accordingly, but these weights can be manually adjusted to fulfill the objective of maximizing predictive accuracy. As THPAM matures in its development, it may be necessary to conceive of an automatic method of detecting feature significance and constructing the generalization matrices with appropriate entry masking or entry flipping level weights.

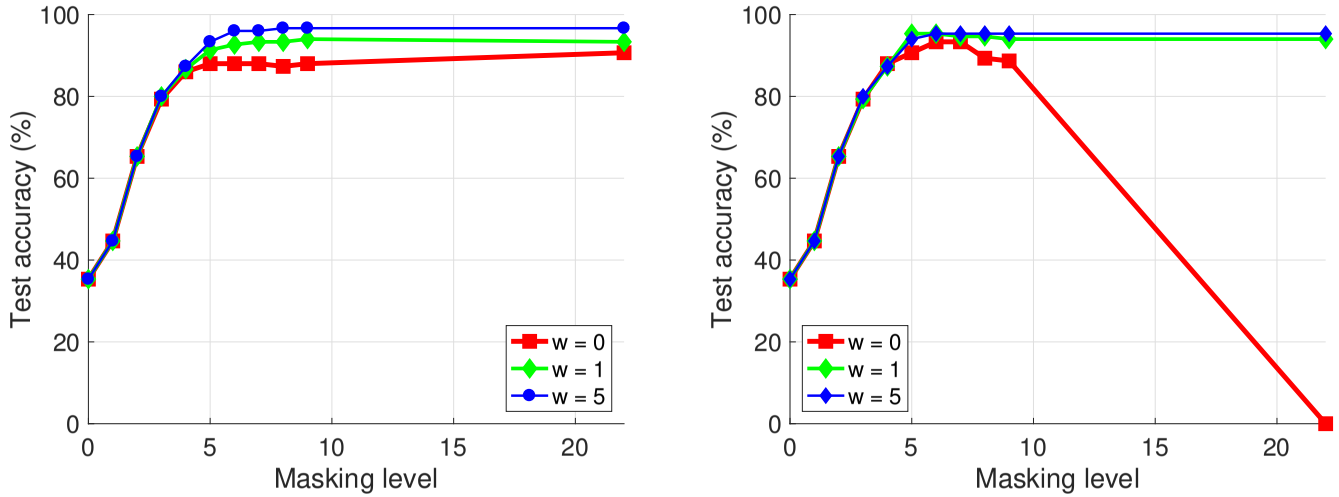
Table 5.17: Leave-one-out cross-validation accuracies on bipolar encoded Iris data by masking level and masking level weight. Quantity w denotes the exponent of the masking level weight 2^{-wj}, where j denotes the number of masked entries from 0 to masking level l.

(a) Accuracy ratings with masking matrix generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=22$
0	35.3333	44.6667	65.3333	79.3333	86.0000	88.0000	88.0000	88.0000	87.3333	88.0000		90.6667
1	35.3333	44.6667	65.3333	80.0000	86.6667	91.3333	92.6667	93.3333	93.3333	94.0000		93.3333
2	35.3333	44.6667	65.3333	80.0000	87.3333	94.0000	94.6667	94.6667	95.3333	95.3333	...	95.3333
3	35.3333	44.6667	65.3333	80.0000	87.3333	94.0000	94.6667	94.6667	95.3333	95.3333		95.3333
4	35.3333	44.6667	65.3333	80.0000	87.3333	93.3333	94.6667	96.0000	96.6667	96.6667		96.6667
5	35.3333	44.6667	65.3333	80.0000	87.3333	93.3333	96.0000	96.0000	96.6667	96.6667		96.6667

(b) Accuracy ratings with entry flipping generalization

w	$l=0$	$l=1$	$l=2$	$l=3$	$l=4$	$l=5$	$l=6$	$l=7$	$l=8$	$l=9$	...	$l=22$
0	35.3333	44.6667	65.3333	79.3333	88.0000	90.6667	93.3333	93.3333	89.3333	88.6667		0.0000
1	35.3333	44.6667	65.3333	79.3333	87.3333	95.3333	95.3333	94.6667	94.6667	94.0000		94.0000
2	35.3333	44.6667	65.3333	80.0000	87.3333	94.0000	95.3333	95.3333	95.3333	94.6667	...	94.6667
3	35.3333	44.6667	65.3333	80.0000	87.3333	94.0000	95.3333	95.3333	95.3333	95.3333		95.3333
4	35.3333	44.6667	65.3333	80.0000	87.3333	94.0000	95.3333	95.3333	95.3333	95.3333		95.3333
5	35.3333	44.6667	65.3333	80.0000	87.3333	94.0000	95.3333	95.3333	95.3333	95.3333		95.3333



(a) Entry masking

(b) Entry flipping

Figure 5.15: Learning curves of a processing unit applied to bipolar encoded Iris data by masking level. Accuracy ratings are shown for quantities $w = 0$, $w = 1$, and $w = 5$ with entry masking generalization (a) and entry flipping generalization (b).

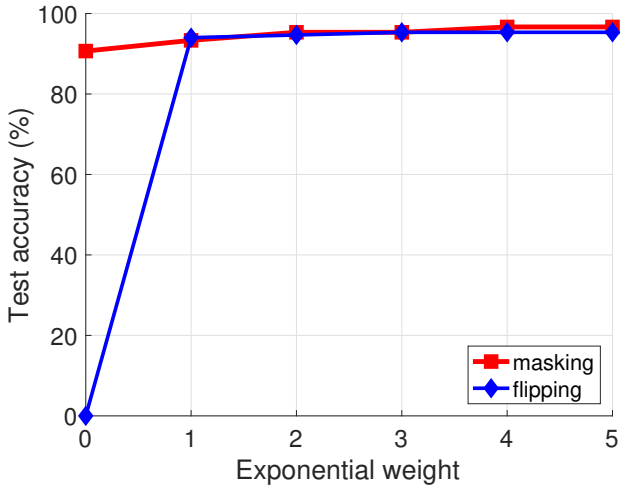


Figure 5.16: Learning curves of a processing unit applied to bipolar encoded Iris data by exponential weight w with masking level $l = 22$. Accuracy ratings are shown for entry masking generalization and entry flipping generalization.

5.4.3 Comparison with Historical Results

In this experiment, stratified 10-fold cross-validation is again used to produce a sample mean accuracy rating with which to compare to known historical results. The WEKA implementations of the J48 decision tree, Naive Bayes, and multilayer perceptron algorithms are reported to achieve 10-fold cross-validation accuracy ratings of 96%, 96%, and 97.33%, respectively [4]. These results are reproducible with WEKA 3.8.1, and the multilayer perceptron result is obtainable with the automatic configuration of 4 input nodes, 3 hidden nodes, and 3 output nodes. The C4.5 decision tree algorithm is reported to achieve 10-fold cross-validation accuracies of 94.33% and 94.67% with boosting and bagging, respectively [11]. There are also re-

ports of 10-fold cross-validation accuracy ratings of 97.333% with a one-against-one support vector machine (SVM) ensemble using the radial basis function (RBF) kernel, 96.667% with a one-against-all SVM ensemble using the RBF kernel, 97.333% with a one-against-one SVM ensemble using the linear kernel, and 96.000% with a one-against-all SVM ensemble using the linear kernel [8].

The processing unit is configured according to the prior experiments examining the influence of the hyperparameters on the observed leave-one-out cross-validation accuracy ratings. The maximum reported leave-one-out cross-validation accuracy is 96.6667% with the bipolar encodings and masking matrix configuration discussed in Section 5.4.2. This is achieved with multiple values of masking level l and exponential weight w, but $l = 22$ and $w = 5$ are selected for this experiment. Figure 5.17 depicts a histogram plot of the 1,000 observed 10-fold cross-validation accuracy ratings. The histogram bars have height proportional to the frequency of the accuracy ratings and width roughly $\frac{2}{3}$ to eliminate gaps present in the accuracy data. Note that of the 150 Iris data instances, an incorrect classification reduces the observed accuracy by 1/150 or roughly 0.6667%, producing gaps in the histogram plot if the bar width is not suitably selected. Table 5.18 summarizes some statistical measures reported in this experiment.

The experimental data demonstrates that the processing unit is capable of performing on-par with several well-known learning algorithms in the current literature. The accuracy rating of 96.6667% occurs with frequency 430 out of the 1000 10-fold cross-validation accuracy samples, and the best reported accuracy rating of 98% compares well with the prior historical results. On a machine running macOS

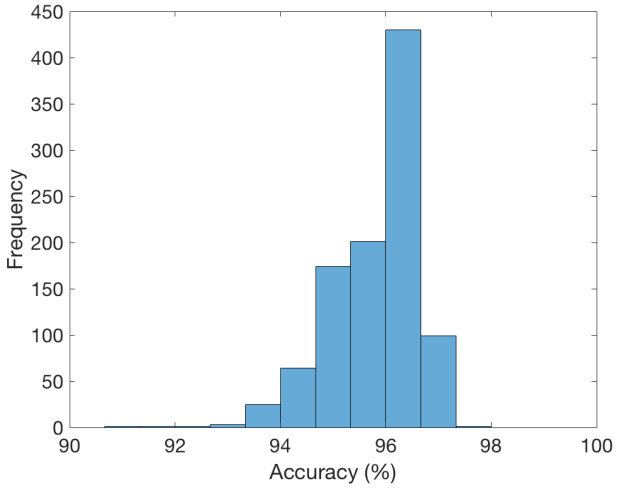


Figure 5.17: Histogram plot of the 1,000 stratified 10-fold cross-validation accuracy ratings of the THPAM processing unit applied to the Iris data set. The bar heights are proportional to the frequency of the observed accuracies, and the bar widths are roughly $\frac{2}{3}$. Masking generalization is used with according to the weighting scheme discussed in Section 5.4.2 with $l = 22$ and $w = 5$.

version 10.12.4 with 16GB 1866 MHz PC3-14900 LPDDR3 memory and a 2.0 GHz dual-core Intel Core i5 Skylake (6360U) with 4 MB L3 cache, the default WEKA version 3.8.1 implementation of the multilayer perceptron performs 500 iterations to learn the entire training set over 0.08 seconds. A serial implementation of THPAM in C running on the same hardware requires only a single iteration to learn the data set over 1.27 seconds. This discrepancy in the observed run times between the multilayer perceptron and the processing unit can be explained by the fact that the multilayer perceptron is capable of processing the four real-valued inputs directly rather than having to encode them as 22-dimensional bipolar vectors. There certainly is room for improvement in the overall mean accuracy rating observed with the processing unit, and there is yet a strong method of improving this further in

Table 5.18: Summary statistics of the 1,000 stratified 10-fold cross-validation accuracy ratings of the THPAM processing unit applied to the Iris data set. Masking generalization is used with according to the weighting scheme discussed in Section 5.4.2 with $l = 22$ and $w = 5$.

Statistic	Accuracy (%)
Mean	96.1493 ± 0.8431
Maximum	98.0000
Minimum	91.3333
Mode	96.6667

future work. Recall that the masking level weights in the masking matrix construction were manually customized so that the relative significance between the bipolar encoded input entries could be better accounted for. Since this was performed manually over few options, it is likely that these weights are not optimal despite the considerable improvements in the observed leave-one-out cross-validation accuracy ratings. An automatic method of determining these weights over a greater range of values would yield further performance improvement on the Iris data set and in other applications.

5.5 Summary

The THPAM processing unit implementation was applied to sample data sets from the UCI Machine Learning Repository in order to examine its performance on various data types and with several data encoding methods. On categorical data, it was demonstrated that the processing unit achieves optimal predictive performance when the feature categories are encoded in a manner that reflects their implied or-

dering in their ranking via Hamming distances, if any such ranking exists. If the data set contains missing values, the bipolar requirement on the input should be relaxed to allow the missing values to be represented with zeros. If data or data encodings contain features which are more significant than others, generalization performance can be improved by manually adjusting the masking level weights so that the masking of these features is discouraged. Generally, the entry masking and entry flipping mechanisms are able to achieve comparable performance providing the exponential weight parameter is sufficiently large. It was also observed that maximum predictive accuracy can be attained even when the masking level is selected to be the input dimension, again as long as the exponential weight is large enough to counterbalance the presence of higher level entry maskings.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Backpropagation based artificial neural networks (ANN), namely multilayer perceptrons, convolutional neural networks, and deep belief networks, are at the forefront of machine learning research and application, serving reliably for many purposes of pattern recognition, classification, function approximation, and signal processing. However, these ANNs suffer from such shortcomings as difficult training, lack of unsupervised learning ability, and ineffective online learning. The backpropagation based ANNs loosely imitate functional clusters of biological neurons, but biological plausibility is not strictly adhered to in the structure of ANNs. Temporal hierarchical probabilistic associative memory (THPAM) is a functional model of biological neural networks that is intended to develop into a learning machine for overcoming these shortcomings.

This dissertation involves the supplemental development and examination of the processing unit as a learning machine. The entrywise product between ternary vectors is demonstrated to be equivalent to the entrywise product of their orthogonal expansions. This enables a new method of constructing masking matrices and an alternative generalization method involving entry flipping. The entry flipping mechanism also can be applied in correlation learning to learn data clusters centered at

a target input, and it can be used to equate the empirical probability formula to a weighted sum which lacks the presence of orthogonal expansions. This may be useful for simulating the performance of a processing unit on data sets consisting of large input sizes, but it is unusable for online learning.

A parallel programming implementation of the processing unit architecture is proposed in this work. The processing unit mechanisms largely rely on matrix algebra and the orthogonal expansion representations of input vectors. Matrix algebra computations are readily parallelizable with introductory knowledge of parallel computing, but the orthogonal expansion implementation was unable to be entirely parallelized. Scalability studies are reported herein, examining the speedup and efficiency of the parallel implementation for fixed input sizes as more parallel processes are utilized. The studies examine the performance of the parallel program when running correlation learning and producing empirical probabilities on 10,000 arbitrary data instances. The program achieves suboptimal speedup and efficiency due to the unparallelized portion of the orthogonal expansion, but the use of parallel programming remains vital in reducing the run times from several hours to a few minutes for sufficiently large problem sizes.

The predictive performance of the processing unit implementation is examined on sample data sets obtained from the UCI Machine Learning Repository. Each data set consists of different data types, including categorical, binary, and real-valued data, with which several encoding methods are considered to adequately represent the data in the form of bipolar vectors suitable for the processing unit. It is evident that Hamming distance relations between the encoded feature values

significantly influence the predictive performance of the processing unit. Input data should be encoded to reflect the natural ordering that exists between the feature values, whether or not such an ordering is present. Cases of missing data entries can be encoded by relaxing the bipolar restriction on the input vectors so that the missing data may be encoded with zeros, a natural extension of the utilization of zeros in entry masking generalization. When data features are disproportionally significant in determining the class label, it is apparent that the generalization mechanisms are unable to detect such significance. The weights of the generalization matrices can be manually adjusted to reflect the importance of particular features over others, leading to greater predictive accuracy.

6.2 Future Work

There are a few processing unit mechanisms and applications proposed in the original publications which remain unexamined in this work. Processing units may not produce the orthogonal expansion of the entire input, but rather they may produce several orthogonal expansions on randomly selected subsets of the input. This would alleviate the curse of dimensionality imposed by the orthogonal expansion algorithmic complexity, so this construction should be examined and, if need be, further developed to perform comparably with the results included in this work. It is also proposed that the processing unit may perform unsupervised learning by constructing pseudorandom output labels as learning is performed. This must especially be critically examined and developed in order to fully realize THPAM

as a recurrent hierarchical network of processing units. Processing units connected to the input are assigned to perform unsupervised learning on subsets of the input data, and the processing unit outputs subsequently serve as input in a new layer of processing units. This structure is theorized to detect and recognize objects and shapes within images, and it could perform dimensionality reduction between layers, enabling its application on more practical and interesting data sets.

Bibliography

[1] Ljupco Todorovski Bernard Zenko and Saso Dzeroski. A comparison of stacking with meta decision trees to bagging, boosting, and stacking with other methods. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 669–670. IEEE, 2001.

[2] Marko Bohanec and Vladislav Rajkovic. Knowledge acquisition and explanation for multi-attribute decision making. In *8th Intl Workshop on Expert Systems and their Applications*, pages 59–78, 1988.

[3] Mark A. Hall Eibe Frank and Ian H. Witten. *Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 4 edition, 2016.

[4] Selma Ayşe Özel Esra Mahsereci Karabulut and Turgay Ibrikci. A comparative study on the effect of feature selection on classification accuracy. *Procedia Technology*, 1:323–327, 2012.

[5] Ronald A. Fisher. The use of multiple measurements in taxonomy problems. *Annual Eugenics*, vol. 7:179–188, 1936.

[6] Matthias K. Gobbert. Parallel performance studies for an elliptic test problem. Technical Report Technical Report HPCF-2008-1, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2008.

[7] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis, 2002.

[8] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multi-class support vector machines.

[9] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on maya 2013. Technical Report Technical Report HPCF-2014-6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.

[10] Samuel Khuvis and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster maya. Technical Report Technical Report HPCF-2015-6, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2015.

[11] Sotiris B. Kotsiantis and Panayiotis E. Pintelas. Logitboost of simple bayesian classifier. *Informatica (Slovenia)*, 29:53, 2005.

[12] Moshe Lichman. UCI machine learning repository, 2013.

[13] James Ting-Ho Lo. Functional model of biological neural networks. *Cognitive Neurodynamics*, 4-4:295–313, Published online 20 April 2010, 2010.

[14] James Ting-Ho Lo. A low-order model of biological neural networks. *Neural Computation*, 23-10:2626–2682, 2011.

[15] James Ting-Ho Lo. A cortex-like learning machine for temporal hierarchical pattern clustering, detection, and recognition. *Neurocomputing*, 78:89–103, 2012.

[16] James Ting-Ho Lo. Personal communication, 2014.

[17] Geoffrey Holmes Bernhard Pfahringer Peter Reutemann Mark Hall, Eibe Frank and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[18] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[19] Andrew M. Raim and Matthias K. Gobbert. Parallel performance studies for an elliptic test problem on the cluster tara. Technical Report Technical Report HPCF-2010-2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.

[20] J. M. Kassim E. H. Gharayebeh S. Makki, A. Mustapha and M. Alhazmi. Employing neural network and naive bayesian classifier in mining data for car evaluation. In *Proc. ICGST AIML-11 Conference*, pages 113–119, 2011.

[21] Jeff Schlimmer. *Concept acquisition through representational adjustment*. PhD thesis, Department of Information and Computer Science, University of California, 1987.

[22] David Slepian. A class of binary signaling alphabets. *Bell Systems Technical Journal*, 35:203, 1956.

[23] Peter J. Tan and David L. Dowe. Mml inference of decision graphs with multiway joins and dynamic attributes. In *Lecture Notes in Artificial Intelligence (LNAI) 2903*, pages 269–281. Springer, December 2003.